ENHANCING MANAGEABILITY OF EXECUTION AND DATA FOR GPGPU COMPUTING

A Thesis
Presented to
The Academic Faculty

by

Anshuman Goswami

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
May 2017

ENHANCING MANAGEABILITY OF EXECUTION AND DATA FOR GPGPU COMPUTING

Approved by:

Dr. Matthew Wolf, Committee Chair
Computer Science and Mathematics Division
Oak Ridge National Laboratory

Prof. Karsten Schwan, Advisor
School of Computer Science
Georgia Institute of Technology

Prof. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Prof. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Prof. Ling Liu
School of Computer Science
Georgia Institute of Technology

Prof. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: December 9, 2016

To Susmita.

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere gratitude and appreciation to my adviser Prof Karsten Schwan. His trust in me was the most important factor leading to the completion of this thesis. I sorely miss his presence today but will remember him always to be the perfect guide for me, as a student. Next, I would like to thank Dr Matthew Wolf, my co-adviser and committee chair, for his immense support in guiding me through the tumultuous phase after the passing away of Professor Schwan and help me reach the end of my PhD journey without giving up. His technical inputs as well as his genuine concern in the forward progress of my academic and professional career is something that I will remain ever indebted for. Besides, I want to thank Dr Jeffrey Young, Dr Greg Eisenhauer and Prof Ada Gavrilovska for their immensely useful feedback in shaping much of the research that has gone into this thesis.

Next, I would like to thank the members of my thesis committee, Prof Ling Liu, Prof Sudhakar Yalamanchili, Prof Richard Vuduc and Prof Hyesoon Kim, for sparing their valuable time and serve as the committee members. The constructive feedback and comments they have provided have gone a long way into refining the content of the thesis as well as stimulate future research directions worth pursuing. I must mention the support and guidance I have received from Prof Venkateswaran as the PhD program coordinator whom I could go to for advice under any situation. I should also mention that the contribution of this thesis has been significantly influenced by my experiences at three different internships for which I would like to thank my mentors, Dr Zhikui Wang and Dr Antonio Lain, at Hewlett Packard Labs; Jonathan Pearce at Intel; and Dr Inam Rahman at Apple.

During my time at Georgia Tech, I was fortunate to have made some very good friends who have become a part of my life. I would like to mention Chang-chih Chen, Pranith Kumar, Nagesh Lakshminarayana, Mohammad Hosaain, Dushmanta Mohapatra, Minsung Jang, Jai Dayal, Alexander Merritt and Dipanjan Sengupta among my past and present labmates who have made this journey so much more enojoyable. A special mention for my

undergrad friends from JU, Ayan Paul and Sudipto Dolui, who have been the ever available support system that I could fall back on during every crisis. Any success would be as much a joy for them as it would be to me.

Finally, I would like to thank my wife for her constant support and unwavering patience through every high and low during my PhD. I have always admired her level of enthusiasm to see me succeed and this has been a key driving force for me to continue striving whatever the situation be. I would like to dedicate this thesis to her without whom I could not have come this far.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

With hardware accelerators like GPUs becoming increasingly common in high end scientific computing environment, new opportunities arise to co-locate scientific simulations and online analysis performed on the scientific data generated by the simulations. However, the offload-driven nature of scheduling of kernels on GPU and the limited context-switching capabilities on the GPU pose challenges to co-locating all the desired tasks.

Additionally, many new cloud-focused applications such as deep learning and graph analytics have started to rely on the high computing throughput of GPUs, but cloud providers cannot currently support fine-grained time-sharing on GPUs to enable multi-tenancy for these types of applications. Instead, scheduling is performed by the GPU driver in combination with a hardware thread dispatcher to maximize utilization. However, when multiple applications with contrasting kernel running times and high-utilization of the GPU need to be co-located, this approach unduly favors one or more of the applications at the expense of others.

This work brings together these two use cases to show how agile coscheduling of separate tasks can be achieved. Through a careful orchestration of compute, memory and network bandwidth resources, the techniques presented in this thesis demonstrate that it is possible to high performance and multi-tenancy in a number of usage and hardware scenarios.

CHAPTER I

INTRODUCTION

A decade ago[91], the Compute Unified Device Architecture (CUDA) API[89] was announced and Nvidia came out shortly after with compatible GPUs and software to support computing on GPUs. Abruptly, this shifted many assumptions about the relative bottlenecks between FLOPs and memory accesses. Other GPU vendors followed suit[63] by adapting their hardware and providing software to enable their GPUs for computing that extends much beyond the domain of graphics and media. Over these ten years, many classes of applications have been adopted to leverage the immense computing potential of GPUs. The HPC community, with a focus on scientific computation, were the early adopters of GPU computing. In recent times, enterprise computing has experienced a sudden burst of GPU use with the growth of computationally intensive analytics in today's era of the big data. Even so, there still exists some fundamental technical and macro-economic barriers in extending modern operating systems, which have evolved based on traditional CPU architectures, to manage GPUs. In particular, the persistent divide between regular schedulable cores and "device" management of GPUs poses issues for users and developers alike. Clearly, there exist opportunites for further pushing the envelope in terms of the performance perceived by the applications while enforcing superior resource utilization. This thesis introduces four different scenarios where there exist opportunities to push the state-of-the-art in GPU computing along with the associated challenges involved, which we detail in the following subsections.

1.1 Managing performance isolation

Accelerators, particularly GPUs, have become the dominant computational engines of today's high-end machines, and this has been recognized and is being exploited by a wide variety of scientific applications, including adaptation of existing ones such as LAMMPS([111]) and GTC-P([137]) and evolution of new ones, such as, PIConGPU([22]). This changing

landscape creates new possibilities for running in situ analysis using heterogeneous compute node resources like GPUs. Such in situ workflows are of significant interest as applications and middleware developers look towards exascale computing, as it offers a way to deal with the significant imbalance between compute capacity and I/O bandwidth by reducing and refactoring data before it exits the machine. Sharing GPUs between long-running scientific simulations and in situ analysis, however, must ensure performance isolation for the execution of the scientific simulations. For simulations that offload some or all of their computation tasks to GPUs, stealing idle GPU cycles presents a different set of challenges. For CPUs, when the co-running analysis task do not completely fit inside gaps, their rapid preemption immediately returns resources back to the simulation process([143]). But such is not the case for GPUs. This makes performance isolation much more challenging with GPUs restricting context switches to kernel boundaries.

1.2 Enforcing predictable sharing

GPUs are being used for more data-rich applications in cloud computing environments than ever before ([81], [134]), and kernels in these applications have increasingly become longer in terms of running time due to larger and larger input datasets. However, in a cloud computing environment like Amazon's EC2[9], long-running kernels on GPUs can hinder scheduling flexibility and fairness for multiple-tenant scheduling. This lack of scheduling granularity and flexibility for discrete GPUs drives up the cost of using GPUs in a cloud computing environment due to limited multi-tenancy and subsequent skews in fair sharing. It also doesn't allow for fair usage of GPU resources that can currently be guaranteed for CPU-based virtualized systems.

All this points to the need for more fine-grained context-switching of GPU kernels that can be done in a manner that is both transparent to the user and improves fairness between multi-tenant kernels. A hybrid scheduling approach is called for which is software-based for managing long-running kernels delegating short-running kernels to the existing hardware scheduler for ensuring utilization remains high.

1.3 Supervizing low-level scheduling

HPC environments, where in situ analysis of data generated from scientific applications provide significant advantages, expose the limitations of the GPU hardware scheduler[10] that is responsible for the thread blocks of compute kernels running on the GPU. The GPU's native assignment policies for thread blocks in several scenarios are not suitable for the execution needs of all components in the overall worklow. As such, valuable scheduling opportunities may be lost. One approach would be to only allow analysis to be scheduled in the gaps of the main scientific application's use of the GPU. Being left only to using the gaps, light-weight, scalable data analysis tasks may not get the necessary amount of GPU cycles to make desired progress.

To address the above challenges, fine-grained and flexible scheduling control of execution on the GPU is necessary to effectively manage in situ analysis of scientific worklows on GPU supercomputers. Software supervized scheduling can combine the existing hardware-based thread block scheduling mechanism with traditional software synchronization-based multi-core CPU scheduling techniques to balance between the conflicting needs of good performance for data-parallel GPU kernels and multi-tenancy on the GPU.

1.4 Enabling distributed computing

Finally we address an emerging opportunity due to changes in hardware. Future supercomputers ([106], [75]) as well as data center clusters[87] with GPUs are likely going to have "dense" configurations where each node will have multiple GPUs with a fewer total number of nodes in the cluster. The majority of workloads for such systems would rely primarily on the GPUs for their computing needs. Examples of applications ready for this transition are ones like fully GPU-based plasma simulations[22] or neural network training for advanced analytics[88]. The on-node bandwidth available for data transfers between GPUs is already much higher with PCIe compared to the cross-node bandwidth from commodity Ethernet switches. For specialized networking hardware like Infiniband, the gap is less. However, the difference is set to widen with latest interconnect technologies like NVLink[95]. So, transferring data between nodes is likely to remain a very significant challenge to enable

distributed GPU computing compared to data movement inside a node.

As such, the network bandwidth of dense GPU clusters is going to be the scarce resource that needs to be carefully managed to maximize the overall performance achieved as well as meet the individual needs of user applications. Traditional network scheduling algorithms from the software-defined networking community, like [6], are geared towards maximizing the bisection bandwidth obtained from the network. However, such an optimization objective may fail to consider the criticality of bandwidth allocation perceived from an application's standpoint. Taking inspiration from the coflow[25] scheduling paradigm proposed for CPU-centric distributed workloads like Spark[141] and Pregel[76], an equivalent approach is necessary to effectively schedule network traffic across co-running distributed applications that rely primarily on the GPU for their computing requirements.

1.5 Thesis Statement

GPUs have become the primary computational engine in supercomputers ([105] and [104]) and datacenter machines ([87]) that house discrete GPU cards. Individual kernels of applications running on such systems are usually hand-tuned to extract maximum performance from the GPU hardware. However, there exists pressing need for mechanisms to co-run kernels from multiple applications on the GPU. On one hand, it enables in situ analysis of data generated from scientific simulations running on supercomputers. On the other, it reduces cost to run long tasks like neural network training on datacenter GPUs through fair-sharing. Moreover, distributed GPU computing over bandwidth-constrained clusters is going to define the next wave of GPU computing riding on machine learning. To address the above needs, this thesis proposes novel mechanisms to achieve effective multi-tenant scheduling of kernels on the GPU as well as application-aware network bandwidth sharing responsive to GPU use. The proposed mechanisms and policies provide flexibility to address different performance goals as well as meet the target levels of resource utilization.

1.6 Contributions

The key contributions of the thesis are the testing and evaluation of set of mechanisms and policies to address the aforementioned gaps in performance and functionality of the

state-of-the-art in management of GPUs. Specifically, we make the following contributions to validate our thesis.

- **Landrush.** We present a detailed analysis of GPU utilization of multiple state-of-the-art science workflows to demonstrate the opportunities that exist for in situ analysis. We design and implement a low-overhead runtime for scavenging idle cycles on the GPU to run data analysis in situ with scientific simulations resulting in lower time to answer.

- **GPUShare.** We characterize the inability of the hardware scheduler in GPUs to achieve fair-sharing across co-running GPU intensive applications with long runnning kernels, a common scenario in the datacenter environment. We present the design trade-offs for such a system and provide an implementation of a low-overhead fair-share scheduling runtime for co-locating compute-intensive analytics jobs.

- **Symphony.** We investigate the scaling limits of individual kernels of well-known scientific workflows to show that top-of-the-line GPU hardware is often over-provisioned even for highly optimized HPC codes. To exploit such scaling bottlenecks for in situ analysis, we design and implement a CPU-like scheduling library to supervize the fully hardware thread block scheduling of GPUs. As a result, the combined throughput gets improved while incurring minimal overhead during standalone execution.

- **GPUCoflow.** We present a qualitative assessment of execution characteristics of deep learning workloads in a dense GPU computing environment, observing that the efficient sharing of bandwidth would be the most critical consideration to maximize overall throughput of clusters running such workloads. We build a simulator to study scheduling policies for interconnect bandwidth sharing in such an environment demonstrating that "coflow" aware scheduling policies perform much better than application-agnostic fair-sharing policies across different workload mixes. Finally, we present a detailed discussion on the design tradeoffs for implementing such a system.

1.7 Dissertation Structure

The rest of the dissertation is organized as follows.

Chapter 2 presents the design and implementation of the Landrush system along with evaluation of the system to demonstrate its effectiveness in enabling in situ analysis on the GPU with reduced time to answer.

Chapter 3 presents the design and implementation of the GPUShare system to enfore fair-sharing among long-running analytics applications in a GPU cloud environment substantiated by evaluation results to prove improvements in fair sharing as well as the degree of slowdown due to overhead.

Chapter 4 presents the design and implementation of the Symphony system that exploits the scaling limitations of scientific codes on state-of-the-art GPU hardware to further stretch the possibilities for in situ analysis on the GPU. Accompanying evaluation is provided to validate the claim with reduced time to answer at reasonable overhead.

Chapter 5 presents the implementation of GPUCoflowSim, a simulator to study the scheduling policies for controlling network traffic, and use it study application-aware bandwidth sharing policies for distributed deep learning workloads that are effective in improving the overall throughput of the cluster compared to SDN-centric policies that focus on maximizing the bisection bandwidth through the network.

Chapter 6 discusses the wealth of existing research that has preceeded the contributions of ths thesis to show how that has inspired the scientific enquiries into in situ analysis, multi tenancy and bandwidth sharing that this thesis has contributed.

Finally, chapter 7 concludes the thesis spelling out some of the future directions to continue research.

CHAPTER II

RELATED WORK

Previous work such as [40] has demonstrated that in situ analytics are important for large GPU clusters like Titan and can also provide important power savings. In addition, research that investigates causes of low GPU utilization([23], [5]) demonstrates that even well-tuned scientific codes leave resources idle. Past work on contention detection and response mechanisms to mitigate interference when applications share resources have focused on CPU and memory interference ([78], [77], [130] and [131]).

Lack of preemption support in GPUs today, makes it difficult to provide performance isolation guarantees. [116] explores preemption-based scheduling but is limited by the GPU programming model and the hardware, with preemption being only possible at kernel boundaries. Pai ei al.[108], building on mcuda[126], provide software-controlled elasticity in the physical resources consumed by a kernel enabling consolidation of kernels that alone do not fully utilize the available SMs on a GPU. Similar ideas were proposed by [19], [45] and [68]. Scientific and production-ready dataceneter applications are already well-tuned to fully utilize the GPU hardware are not likely to benefit from such mechanisms.

Concerning software stacks, the earliest entirely user-level solutions were gvim[46], vcuda[121], gvirtus[41], and rcuda[38]. More recent work([118], [110], [16], [113]) including vendor-supported libraries[97] have tried to exploit hardware improvements such as multiple hardware queues to improve utilization. All of these mechanisms rely on scheduling actions after kernels have already run limiting their effectiveness for long-running kernels. Other solutions, RGEM[60], timegraph[61], gdev[62], Basaran et al.[15], Menychtas et al.[82], Tian et al. [132] and gpuvm[127], replace the vendor-provided software stacks in order to promote GPUs to first-class schedulable entities in the OS and/or hypervisor, but unfortunately they also rely on reactive scheduling actions similar to other user-level middleware.

Hardware solutions[4] to support spatial multitasking on GPUs have been proposed.

Two preemption mechanisms, namely, *context switching* and *SM draining*, are proposed in Tanasic et al. [129]. *Context switching* can provide response-time guarantees, but has prohibitive overhead. *SM draining* does not dispatch new thread blocks and preempts only when the currently executing ones complete. Chimera [109] builds on the above by combining the low latency of *context switching* with the high utilization of *SM draining*, into a single mechanism called *SM flushing*. Assumptions are that kernels are idempotent and if not, they distinguish between global memory writes and overwrites. As hardware changes were proposed, evaluations were performed on simulators[13], making it difficult to gauge if real application kernels would benefit from such mechanisms.

PTask[114], Dandelion[115] and GEMTC[66] offer alternative programming models that give the OS sufficient visibility about an application's use of the GPU, thus permitting it to provide isolation and fairness guarantees. Moving existing GPU applications to such new models would require significant effort. Low-level GPU-side profiling tools are available now like SASSI ([125]) that enable high fidelity GPU code instrumentation. Some of the contributions in this thesis have explored profile-guided execution on the GPU.

The fundamental premise of many of the contributions of this thesis is based on the seminal work on realtime scheduling by Liu and Layland[74]. There has been a good deal of work focused on scheduling for a hybrid system of CPUs, GPUs, [73] and other accelerators [17] but little work on coscheduling applications within a set of GPU accelerators. Both AMD and NVIDIA support hardware virtualization and have projects to support varying levels of user scheduling, but most of this development is focused on improving the performance of graphics-intensive workloads. Previous work on CPU "core pinning" techniques [65] are also relevant.

The parameter server approach for distributed machine learning[71] has been shown to scale well across thousands of nodes with multi-core CPUs. But, artificial neural network training were not among the algorithms evaluated. Given that ANN training is characterized by higher FLOPs to bytes ratio, their performance sensitivity to the use of network resources needs careful consideration. Large-scale deep learning on thousands of machines[31] using the CPU cores have been shown to work. But there is growing evidence[87] that they can

be matched by the processing power of merely tens of machines where each machine hosts multiple GPUs.

Recent work[28] has shown distributed deep learning using GPUs can achieve very good speedup. While speedup of individual training jobs are definitely important, overall throughput from a cluster shared by many such jobs is becoming critical as most of these jobs are going to run on shared infrastructure. Other work on intra-node bandwidth sharing over the PCIe[79] (or in future, NVLink) complement some of the contributions in this thesis.

Coflow scheduling([25],[26],[142]) has attracted a lot of research in the last few years due to growing prominence of several big data frameworks([32],[141],[76]) that could benefit from it. MPI has been the distributed programming model of choice for the scientific community and CUDA-aware[93] versions have been around for a while. Whatever be the communication API, coflow aware scheduling holds great promise for distributed GPU computing.

CHAPTER III

LANDRUSH : RETHINKING IN SITU ANALYSIS FOR GPGPU WORKFLOWS

The evolution of high-end computing has allowed scientific simulations to run at higher and higher fidelity, which in turn generates massive amounts of data. This makes infeasible the traditional approach to I/O in which data is written to disk for subsequent post-hoc analysis. Running analysis where and when data is generated, often referred as "in situ" analytics', becomes a necessity rather than an option, particularly for the exascale era [37].

Previous work ([3], [2], [144], [143], [18], [133]) has demonstrated and explored a wide range of solutions for in situ data analysis, ranging from early work on "data staging" to recent work in which scientific simulation and analysis tasks efficiently time-share the CPU resources of compute nodes.

An issue not considered in such prior work is the rapid evolution of node architectures from previously homogeneous, CPU-only platforms to richly heterogeneous machines with accelerators and complex memory hierarchies. Accelerators, particularly GPUs, have become the dominant computational engines of today's high-end machines, and this has been recognized and is being exploited by a wide variety of scientific applications, including adaptation of existing ones such as LAMMPS [111] and GTC-P [137], [54] and evolution of new ones, such as, PIConGPU [22].

This changing landscape creates new possibilities for running in situ analysis using heterogeneous compute node resources like GPUs. Sharing GPUs between long-running scientific simulations and in situ analysis, however, must ensure performance isolation for the execution of the scientific simulations. [143] showed that CPU sharing on compute nodes can be done inobtrusively and effectively, by stealing cycles during the serial phases (MPI communication and I/O) of scientific simulations. However, for simulations that offload some or all of their computation tasks to GPUs, stealing idle GPU cycles presents

a different set of challenges. Programming models for today's high-end discrete GPUs are specialized to utilize their high numbers of processing cores and to deal with the latency of offloading over the PCIe bus. Consequently, the task of identifying idle cycles on the GPU has to account for availability of both compute and data transfer resources. Even when such idle gaps are identified, the penalties for mis-predicting them is higher on GPUs than CPUs. This is because on CPUs, an analysis process may run during the idle gaps created by serial phases in the simulation process. If the co-running analysis task does not completely fit in the gap, its rapid preemption immediately returns resources back to the simulation process. In contrast, GPUs restrict context-switch to kernel (parallel programs running on GPUs) boundaries which is a far cry from the instruction-level precise interruption that system designers are used to in the CPU world, thus making performance isolation a much more challenging task([116], [129]).

In this work, we seek to unlock the GPU resources present on high-end machines for use in running in situ analysis. The advantage of such resource sharing is multi-fold, including (i) the ability to operate on scientific data without moving it to staging nodes thereby reducing network traffic and energy cost, (ii) the accelerated execution of analysis tasks on energy-efficient hardware accelerators, and (iii) the efficient use of all resources present on compute nodes – CPUs and GPUs – to run scientific applications and associated analysis routines.

To this end, we have designed a runtime system, termed *Landrush* [1], for enabling GPU sharing while also ensuring that GPU sharing does not unduly perturb a scientific simulation's execution. The Landrush solution makes the following technical contributions:

- We characterize the GPU activities for three different types of high-end scientific simulations, demonstrating the feasibility of task co-location using spare GPU cycles.
- We show a reduction in the total 'time-to-answer' seen by end users, by recognizing GPU idle gaps for executing analysis tasks.
- We study the effects of co-locating different complexities of analysis tasks with various

[1] 'Landrush' is the chaos that ensues from opening previously restricted territory to new uses.

scientific simulations, leading to a better understanding of the GPU sharing characteristics of (simulation, analysis) pairs.

The Landrush approach is evaluated with three state-of-the-art scientific simulation applications, LAMMPS, GTC-P, and PIConGPU, on the Titan [105] supercomputer hosted at Oak Ridge National Laboratory (ORNL). Experimental results demonstrate that up to 90% (!) of the GPU cycles are available over timesteps spanning 100-200 ms that can be used for analysis task co-location. Compared to naive co-location relying on scheduling by the GPU driver and the hardwired thread dispatcher, reductions in time-to-answer range from 8% to 33% across different combinations of co-location with nearly uniform performance from very small (4 nodes) up to very large scales (1024 nodes).

3.1 Motivation

We seek opportunities for sharing the GPU resources on high-end machines between scientific applications and representative analysis applications that consume the generated scientific data.

This requires answering the following questions:

1. How much of the GPU is left unused by scientific applications?
2. Why offload analysis computations to GPUs?
3. What about moving data to/from the GPU?
4. How can we restrict analysis tasks' resource usage to idle cycles?

3.1.1 How Much of The GPU Is Left Unused?

A survey of existing HPC applications [102] provides basic information about their use of GPUs. For applications not using the GPU (first row in Table 1), the GPU is idle for the entire duration of the scientific application. For applications using the GPU (second and third rows in Table 1), our analysis results in Section 3.3.1 show considerable variability of GPU idle cycles for a constant problem size of analysis tasks and at the strongest scaled point of simulation. The GPU is idle in LAMMPS (Lennard Jones potential) for 88% and

in GTC-P for 15%. Even an application like PIConGPU, which has been written to perform all of its computations on the GPU, exhibits idle gaps between GPU activities on a subset of the nodes running the application. These gaps result because it is difficult to eliminate all serial phases for applications using the GPUs and also because data movement between nodes cannot be fully overlapped with computation (due to complex transfers from the GPU to the host memory and then to the network device).

Table 1: GPU Use in Representative Scientific Applications

GPU Use	**Applications**
Not used	Tinker, Dirac, RedMD, GTS, Pixie3D and many legacy codes
Partially used	GROMACS, LAMMPS, NAMD, MILC, NWChem, ENZO, GTC-P, CP2K, Octopus
Fully used	HOOMD-Blue, S3D, PIConGPU

3.1.2 Why Offload Analysis Computations to GPUs?

Enabling in situ analysis requires harvesting resources whenever and wherever they are available. [143] exploits the benefit of co-running analysis when idle cycles on CPUs are available. With accelerators like GPUs becoming more prevalent on HPC machines and with end users demanding ever richer online analysis pipelines for processing scientific data, it is imperative to run them on the "fast" resources available on high end machines as opposed to simply delegating them to be run on CPUs. This is particularly the case for computationally intensive codes, i.e., those with small gaps on CPUs or GPUs, for which analysis computation tasks should be run wherever they run fastest.

As evidence, we reference the published performance reports for standard kernels [57], [96] that constitute common analysis tasks, running on CPUs vs. GPUs. Even when compared across state-of-the-art hardware (for both CPUs and GPUs), highly optimized (MKL and TBB for CPUs; cuFFT and Thrust for GPUs) versions of the same algorithms clearly perform best when run on the GPU with problem sizes that are similar to scientific output data. The speedups in computation time range from 2-5x for reduction, sorting and FFT in that order.

3.1.3 What About Moving Data?

A known challenge with using powerful, discrete GPUs (over less powerful, integrated GPUs) is the need to transfer data from system memory to GPU memory over the PCIe link connecting them. Note that this is also a problem for running analysis tasks on idle CPU cores. Our study of high end scientific applications shows that the GPU's DMA engines are not heavily used by the scientific simulation: the percentage of DMA use in LAMMPS (Lennard Jones potential), GTC-P and PIConGPU constitutes only 1.4%, 4.5%, and 0.2% of their running times.

Moreover, programming support like CUDA's inter-process communication (IPC) has further reduced the cost of data movement between memories. IPC allows for sharing buffers between different GPU contexts, and Landrush uses this functionality to efficiently co-locate analysis applications that are node-local and to reduce intra-GPU memory transfers.

3.1.4 How to Restrict Analysis Tasks' GPU Use to Idle Cycles?

Precise interrupts are not supported by the hardware for GPUs. It is difficult to store the context when interrupting more than a hundred concurrently running hardware threads of execution (orders of magnitude higher than the degree of hardware multi-threading in CPUs). Moreover, GPGPU kernels are written to mostly preserve a uniform control flow to make best use of the SIMD execution model. In most situations, there is little to gain from interrupting a GPU thread block (CUDA terminology for a group of threads that share a software managed cache with hardware synchronization) before completion.

However, GPU programming models expose a batching parameter for kernel launches that goes beyond the granularity of execution of a GPU thread block. An application may choose to launch several thread blocks when it makes a kernel launch call. For analysis kernels, it may be the case that the running time of a few (but not all) thread blocks fits inside gaps on the GPU.

In Landrush, analysis kernels are instrumented with an availability check at the start to allow early completion in case a kernel from the scientific application is launched while the analysis kernel is still running. So it is possible to emulate finer-grained software interruption

capability despite lack of hardware support for context switch in GPUs.

3.2 Landrush System Design and Implementation

To enable GPU sharing between co-located scientific simulation and analysis, Landrush functionality includes:

- monitoring GPU calls from the scientific simulation and kernel running times on the GPU for both the simulation and the analysis applications
- informing analysis-side Landrush about opportunities to use (or defer use of) the GPU
- managing kernel launch calls from analysis based on four availability heuristics
- scheduling simulation and analysis kernels using the GPU driver
- restricting analysis' use of compute cycles to within the gaps.

The overall architecture of Landrush is shown in Figure 1 which depicts how the above functionalities interact. They are described in more detail below.

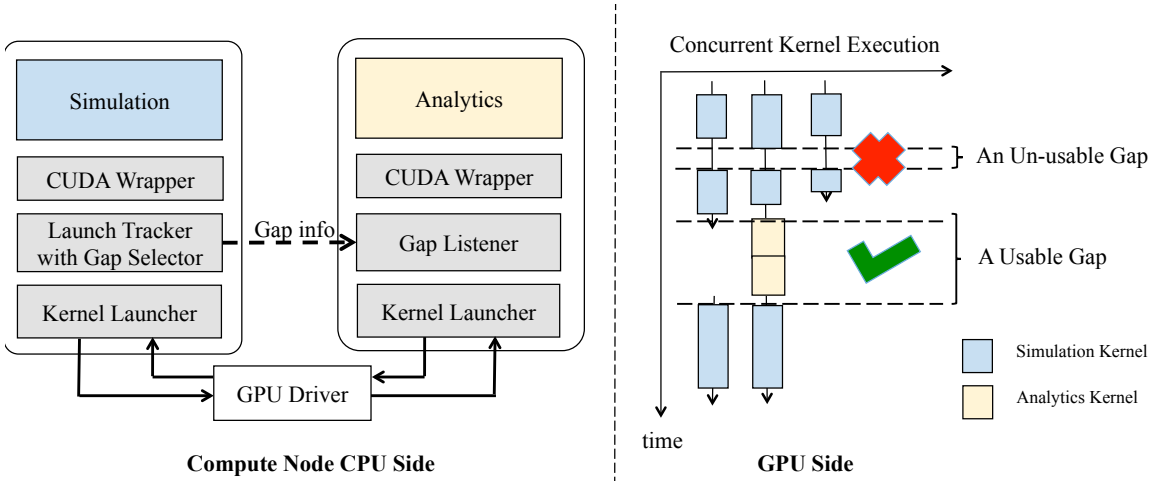

Figure 1: Landrush Components and Gap-Aware Co-Location

3.2.1 Monitoring Usage

Landrush is implemented as a library that resides within the address space of both the science and the analysis applications. It operates by intercepting either's GPU kernel launch

calls via wrapper APIs. Kernel launch calls from both the scientific and the analysis applications are accounted and passed to the GPU driver. Analysis kernel launch calls are passed to the driver directly only if no science kernels are waiting to run, otherwise they are deferred.

To track the start and completion of every single kernel execution on the GPU as they occur, "events" exposed by GPU programming models have to be used. But to do this asynchronously, additional threads have to be run to poll for such events which adds a lot of overhead as the number of events increase. Therefore, Landrush relies on an asynchronous but repeated supply of kernel start and end timestamps by asking the GPU runtime to deliver the necessary profiling information[99].

GPGPU calls made by an application are detected by Landrush as they occur by using a wrapper around the runtime. Besides kernel launch calls, Landrush intercepts all data transfer, event and queue [2] related calls from the scientific application. Some of these calls are synchronous, while others are asynchronous. The profiling information sent from the scientific applications consist of (1) the timestamps of the above GPGPU calls, and (2) the GPU use and idle interval data alongwith the kernel(s) corresponding to each such interval. The GPU runtime delivers (2) asynchronously to Landrush which is then sent to the analysis-side component using a light-weight messaging library called nanomsg[85].

On the analysis side, Landrush only tracks GPU execution start and end intervals (but not idle intervals) alongwith the corresponding analysis kernels. It tries to ensure that the science application receive execution time on the GPU whenever it launches a kernel. It also runs an additional thread on the analysis-side to receive the profiling data which also acts as a listener for messages that indicate upcoming gaps on the GPU to run analysis kernels, described next.

3.2.2 Communicating opportunity triggers

In order for the Landrush component on the analysis side to be aware of gaps on the GPU, there needs to be a constant supply of up-to-date GPGPU interactions made by the

[2]Queue (command queue or stream) is another type of logical object exposed by GPGPU programming models that can be used to increase concurrency of execution on the GPU.

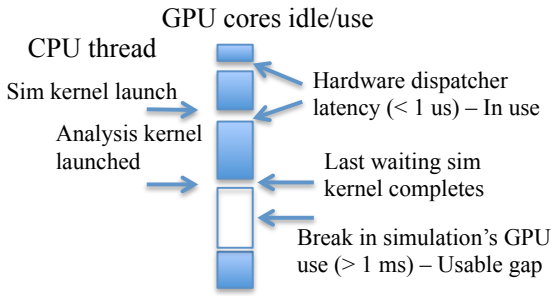

Figure 2: Actions on kernel launch call

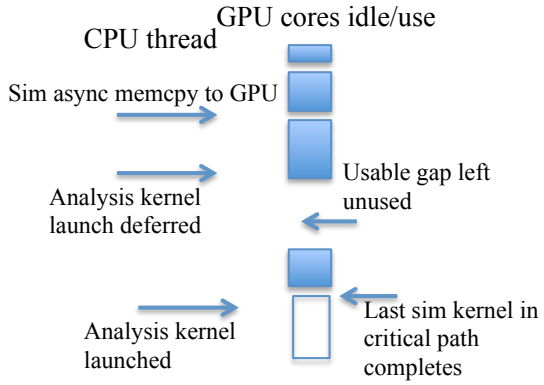

Figure 3: Actions on asynchronous memcpy calls

scientific application. Communication is triggered whenever a GPGPU interaction occurs and is deemed to convey new information. But, this consolidated profiling data is not current but from the recent past. Information about current usage of the GPU by the scientific application is necessary to decide on gaps.

When several simulation kernels are launched in quck succession (no idle time on GPU for a while), sending a message on each such launch can cause an unacceptable slowdown compared to standalone running time. To address this problem, a message is only sent when the last in a batch of kernels are launched or a subsequent data transfer is requested. Such message limiting is enforced by starting a timer (of 1 ms or the context switch interval on the GPU) after every message is send out and then waiting for the timer to expire before sending a new message consolidating all GPGPU interactions that occur during the interval.

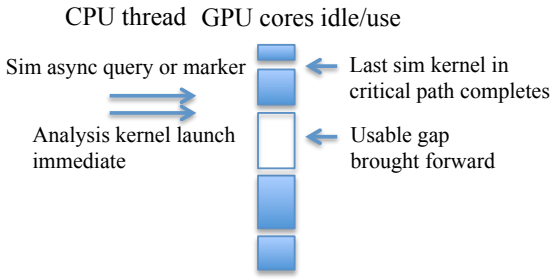

Figure 4: Actions on other asynchronous calls

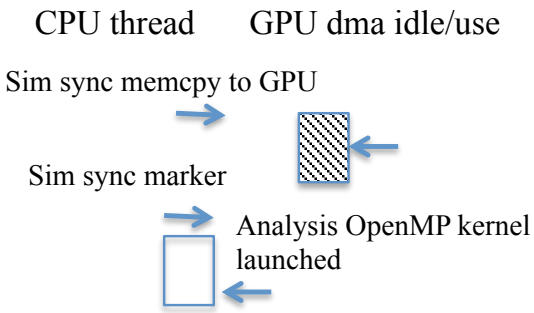

Figure 5: Actions on synchronous calls

3.2.3 Managing analysis' kernel launch calls

A kernel launch from the analysis application is intercepted by Landrush, and a decision algorithm is run to determine gap availability on the GPU. Even if a single gap is not large enough to fit the analysis kernel but the aggregate of multiple gaps are, the GPU is still used. Only if the aggregate of available gaps on the GPU fall short, the analysis kernel is run on the CPU. This is due to the much superior performance (at least faster by 2x) of the analysis kernels when run on the GPU compared to the CPU for the size of output data produced by the studied scientific simulations.

A gap on the GPU is detected if a period of no GPU activity exceeding 1 us is found as that is considered sufficiently larger than the maximum overhead to schedule the next kernel on the GPU if one was available. But, a gap is considered usable if it is estimated to be more than 1 ms, the context switch interval enforced by the GPU driver. On the CPU side, a gap is considered to be an interval of busy-wait or sleep of 100 ms or more, an order of magnitude more than the maximum scheduling latency of the OS. Gaps on CPU are only

considered when the aggregate of usable gaps on the GPU is less than the analysis kernels' total running time as determined from the most recent profiling information.

The decision engine operates on the basis of three heuristics in determining gaps. If the last message is a kernel launch call, it looks up the most recent profiling data identifying the most recent occurrence of this kernel launch and the corresponding usable gap immediately succeeding it. A timer is started to wait for the interval until the next "usable" gap. If a previous record is unavailable, the analysis kernel is allowed to run right after the current kernel completes execution. Otherwise, the analysis kernel is allowed to run after the expiration of the timer as shown in Figure 2.

If the last message is an asynchronous query or GPU-side synchronization (event or queue) or an asynchronous memory copy (GPU to system memory), the decision engine interprets that as an indication that the kernels that need to complete as directed by the last interaction, cannot be deferred. It uses this information to check if the timer expiration could be brought forward by deferring kernels not indicated to be in the critical path. This allows analysis kernels to run before those simulation kernels, as shown in Figure 4. On the other hand, asynchronous memory copy message in the other direction (system to GPU memory) or between buffers on the GPU delays the timer expiration interval by adding the running times of all kernels which are potential consumers of this data movement, as shown in Figure 3.

If gaps on the GPU are not enough as determined from the most recent profiling data, a synchronized memory copy (system to GPU memory or between buffers on the GPU) is used to transfer simulation's output data from the GPU to the system memory. Thereafter, a busy-waiting GPU-side synchronization (on event or queue) is used to run analysis tasks on the CPU. This is shown in Figure 5. Note that this is more generalized than [143] which restricts CPU core use by in situ analysis to never use the core(s) used by the science application. In this case, as the CPU-side application thread is in a busy-waiting or sleep state, it is not necessary to measure interference caused by co-locating an analysis thread on the same core. Synchronous memory copy messages (GPU to system memory), currently unused, presents opportunities to transfer analyzed data back to the GPU memory if a

multi-stage analysis workflow is used.

3.2.4 Multi-context kernel scheduling by the GPU driver

Two different GPU contexts cannot time/space share the GPU. This prevents even low footprint analysis kernels from executing on the GPU concurrently with other low footprint kernels from the scientific simulations. By running microbenchmark kernels busy-wait on the GPU for variable intervals of time, we deconstructed the behavior of the driver when trying to simultaneously run kernels from more than one context (in our case, the scientific and the analysis applications).

One of two things happen when an analysis kernel is launched and the GPU is idle. (1) If no other kernel launched by the scientific simulation is in the queue, the analysis kernel can utilize the GPU before the next simulation kernel is launched. If the analysis kernel completes inside the gap, then co-locating analysis works out to be better than running serially. (2) If there are simulation kernels waiting to run, the driver schedules the kernels on the GPU in a round-robin manner with a context-switch interval of 1 ms or when an executing kernel completes. Multiple kernels from a context gets to run until their cumulative running time exceeds 1 ms, at which point, kernels from another context, if waiting, gets to run. If there are no other kernels waiting from other contexts, the same context can keep using the GPU because of the driver's work-conserving scheduling policy.

The above policy breaks down when execution times are near tens or hundreds of milliseconds (which is the case for problem sizes corresponding to the generated scientific output data). The effective context switch interval is reduced to a long running kernel's execution time and explicit scheduling of analysis kernels (not relying completely on the GPU driver) is needed.

In Landrush, analysis kernels are scheduled inside gaps when kernels from the scientific applications are not using the GPU. At any given time, it is ensured that analysis kernels are passed to the GPU driver only when there are no "kernels in the critical path" from the scientific application already waiting to run.

3.2.5 Enforcing analysis kernels to fit inside gaps

In Landrush, analysis kernels are instrumented to insert an availability flag check that runs at the start of each thread block and determines if the GPU is still available to use. Any thread block only runs to completion, if the GPU is still available. The availability flag is stored in the GPU memory and is updated by the gap availability decision engine. Once one or more thread blocks detect the GPU to be unavailable, the remaining thread blocks return early using mechanisms described in [68]. Since the thread block dispatch is done in hardware, the overhead is negligible even for very large number of thread blocks. The same analysis kernel is next resumed to execute from the first thread block that was skipped in the last launch and allowed to run until the availability check fails, with each "round" completing more of the analysis kernel's overall grid of thread blocks, until it is complete. On the CPU, analysis kernels are run during gaps created by the busy-waiting episodes of the scientific application, and POSIX signals (from the OS) are used to interrupt execution beyond the gaps and revoke resource from the analysis tasks.

3.3 Evaluation

All experimental evaluations are performed on the Titan supercomputer at ORNL. Titan consists of 18,688 compute nodes. Each compute node contains a 16-core 2.2GHz AMD Opteron 6274 processor and 32GB of RAM. Two nodes share a Gemini high-speed interconnect router. The resulting partition contains 299,008 traditional processor cores, and 587TB of memory. Each compute node is also equipped with an NVIDIA Kepler K20 GPU accelerator [92] with 6GB of DDR5 memory. Experiments broadly evaluate two questions:

1. What idle time is available on the GPU for each of the science codes (LAMMPS, PIConGPU, GTC-P), usable for co-running analysis?

2. To what degree does co-running some given analysis with a scientific simulation on the GPU lead to reduction in the total time to answer seen by the end user?

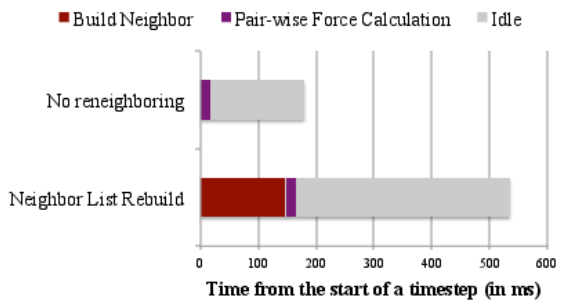

Figure 6: Gaps in one time step of Lammps Lennard Jones Potential

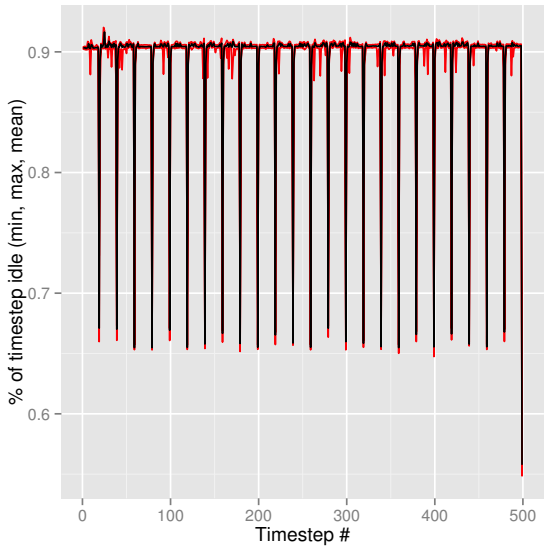

Figure 7: Lammps Lennard Jones Potential gaps across 64 nodes

3.3.1 Idle Period Analysis

Extensive characterizations of the GPU usage of LAMMPS, GTC-P, and PIConGPU demonstrate substantial variability in the extent to which they use the GPU for offloading parallel execution phases, both in terms of occupancy on the GPU and the regularity in their usage patterns.

3.3.1.1 LAMMPS

LAMMPS (or Large-scale Atomic/Molecular Massively Parallel Simulator) is a well-known molecular dynamics simulation with both OpenMP based CPU and CUDA based GPU acceleration. The version of LAMMPS run in our experiments is configured to perform

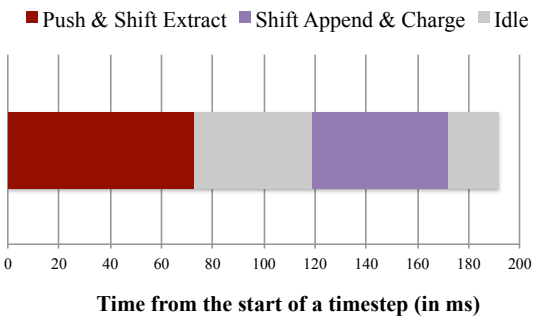

Figure 8: Gaps in one time step of GTC-P

Lennard Jones Potential calculation using GPU acceleration with the core GPU package of LAMMPS (other user contributed GPU packages are available but they are not officially supported by LAMMPS). The input data size used is 2.1 million atoms per GPU requiring 3 gigabytes of memory on a K20 on Titan with a memory capacity of 6 GB. The run configuration consists of 500 steps of dynamics with the velocity-Verlet integrator across 64 nodes (one GPU per node).

Two kernels run on the GPU when LAMMPS is configured with the above parameters: one performs the pair-wise force computations, in each time step, while the other re-calculates neighbors to be used for the force computations, once every twenty steps. For the given input size, the force computation kernel runs for around 17 ms. For 86% of the running time when neighbor calculation is not being performed, the GPU activity constitutes 9% of the average length of a time step. The remaining 14% of the running time are constituted by longer time steps due to the neighbor calculation taking around 146 ms. But even during these steps, the GPU use is only around 33%. This is shown in Figure 6. This is due to fact that the force field computations for Lennard Jones are performed on the CPU. The GPU is idle for nearly 88% of the application's running time. Also, the idle periods appearing in a time step remain nearly constant over the course of the simulation as does their distribution across nodes, shown in Figure 7. To summarize, LAMMPS configured to run Lennard Jones Potential using the GPU, exhibits a low but regular GPU usage.

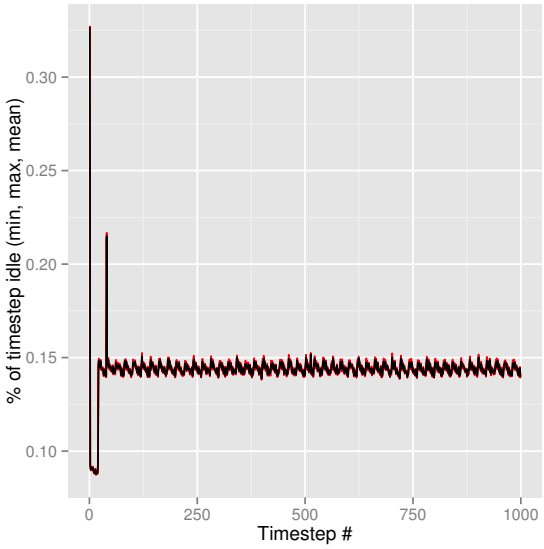

Figure 9: GTC-P gaps across 64 nodes

3.3.1.2 GTC-P

GTC-P (or Gyrokinetic Toroidal Code - Princeton) is a physics simulation evolved from GTC that uses the GPU. It is a solver for the gyrokinetic equation employing a 2-D domain decomposition in the radial and toroidal dimensions respectively. We use a constant toroidal dimension of 4 and vary the radial dimension to achieve weak scaling. The input data size is 12.9 million particles per GPU, resulting in a GPU memory footprint of 2.7 GB which is the highest problem size that can be fit on the K20 GPUs on Titan. GTC-P is run for 500 steps across 64 nodes with each step consisting of 2 sub steps, one for each stage of the Runge-Kutta algorithm that advances simulation time.

There are four main GPU operations that constitute the bulk of GTC-P's GPU usage, viz., push, shift(both toroidal and radial directions) and charge. Some other operations like smoothing, field and poisson calculations are performed on the CPU using OpenMP for parallel phases. As shown in Figure 8, this creates a window of 115 ms, for the above input size, during which the the GPU is left unused by the science application which accounts for nearly 15% of a time step. Moreover, these gaps repeat in every time step and are observed on all nodes running the simulation. This is shown in Figure 9. Thus, GTC-P exhibits high use of the GPU with regularly occurring windows of idleness in each timestep.

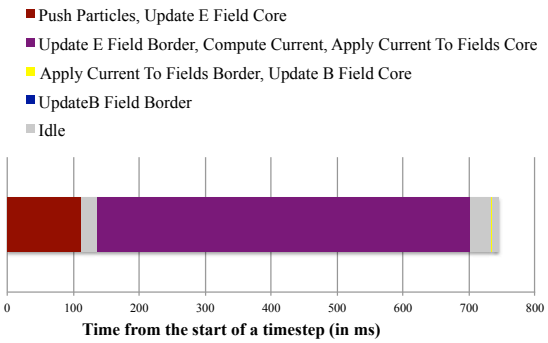

Figure 10: Gaps in one time step of PIConGPU

3.3.1.3 PIConGPU

PIConGPU (or particle-in-cell on GPU) is another physics simulation implementing the PIC (particle-in-cell) method that computes the fully relativistic motion of electrons and ions in the presence of electric and magnetic fields governed by Maxwell's Equations. We configure PIConGPU to simulate a Kelvin-Helmholtz Instability (KHI), where the particles are placed in grids using 44.2 million particles per GPU, requiring a minimum of 2 GB of memory on the GPU. PIConGPU uses a GPU-resident memory allocator, called mallocMC, that pre-allocates most of the available GPU memory and then services runtime allocation requests from GPU kernels. PIConGPU is run for 500 time steps across 64 nodes.

There are four main phases in a time step (shown in Figure 10), each of which run several kernels on the GPU. In the first phase, particles (ions and electrons) are pushed based on the current field, lasting for 15% of the length of a time step. Fields at the border of a grid depend on neighboring grids located on other nodes. So, field towards the center of the grid are updated while waiting for data from other nodes to arrive. The delay in arrival of data from other nodes shows up as the first idle window, idle 1. The current computation phase, is the longest, constituting around 75% of the time step; it is also the busiest with practically no gap on the GPU. The computed current is applied to the fields during the third phase. Similar to the update of fields after particles are pushed to new positions, this phase also involves updating the field based on the newly computed current values. Like before, this also involves waiting for data from other nodes, resulting in, idle

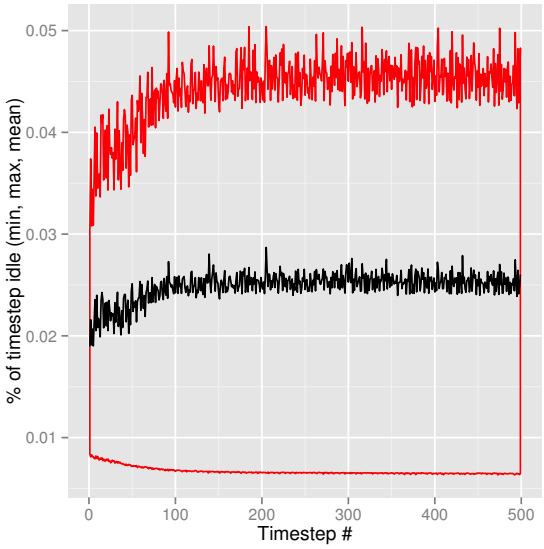

Figure 11: PIConGPU gaps across 64 nodes

2 (nearly 4% in the time step), shown. The first gap, idle 1, only appear when the overlap between push particles and field update does not hide all data dependencies. The second gap, idle2, occurs when the computed currents have been used to update the field towards the center but fields from borders of neighboring grids have not been received yet. As the field data is much less than the particle data, sufficient computation task is not available to hide the communication latency and results in the only usable gap during a time step of PIConGPU.

As shown in Figure 11, there is some variation in the length of this idle period across 64 nodes but the maximum idle period for any node is under 5% of a time step. Landrush in its current form is a local scheduling entity. It would be intersting to look at the utility of multi-node management frameworks [30] in aggregating such idle information and scheduling varying amounts of insitu work based on a node's load, if the data movement cost can be compensated. To sum up, PIConGPU is characterized by very high GPU usage, affording only modest co-location for 0(N) analysis (like scan, minmax, mean etc) on the GPU with the CPU serving as the main available resource for performing more compute-intensive insitu analysis.

3.3.2 Improvement in 'Time to Answer' with Gap-aware Co-location

In this section, we discuss the performance implications if a scientific simulation's GPU idle periods are used to run some analysis codes useful for the produced scientific data. We use three commonly used analysis algorithms run on scientific data: Convolution, Scan and Histogram.

We have shown in the previous sections that different scinetific codes exhibit varying gap amounts on the GPU. We next compare the effects of co-location and serialized execution (Serialized) on the time-to-answer expressed as the % increase in running time over the running time of a standalone execution of a scientific simulation. Two co-location strategies are used: one naively co-locates analysis tasks (Naive) on the GPU relying on the driver and the internal HW threadblock dispatcher, unaware of any contention that may arise; the other (Landrush) co-locates analysis during gaps in the execution of the scientific simulation when the GPU is idle. 100 time steps are executed for the three scientific simulations - LAMMPS, GTC-P, and PIConGPU. The input data size for the analysis application is chosen to be the largest allocated single buffer in the co-running simulation application. Weak scaling is performed at three scales, viz., 4, 64 and 1024 nodes. Due to the limit on atom indices in the Lennard Jones configuration of LAMMPS, the highest scale was limited to 512 nodes.

3.3.2.1 Convolution

Convolution is a fundamental analysis subroutine used by several other higher-level analysis algorithms, most notably, FFT or Fast Fourier Transform. We used a 2-dimensional convolution algorithm where the x- and the y- dimensions are separable using an intermediate buffer. Two kernels are run one for each dimension. The filter co-efficients are stored in constant memory on the GPU to enable fast access.

Experimental results with running convolution in situ are shown in Figures 12, 13 and 14. At a weak scaling of 4 nodes, both serialized and naive configurations co-locate really badly causing slowdowns between 23 and 61%. For the same configuration, the highest slowdown with Landrush is less than 3%.

At 64 and 1024 nodes, the slowdown becomes more tolerable for GTC-P and PIConGPU

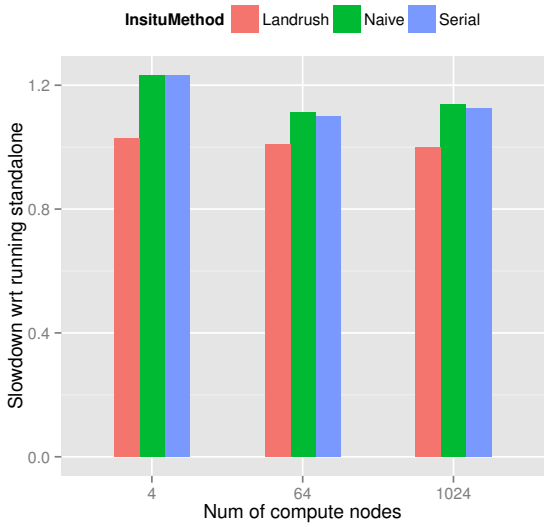

Figure 12: Co-location Performance with Convolution : GTC-P

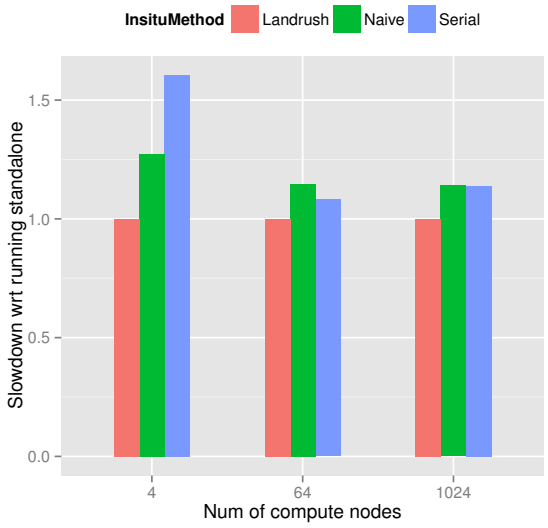

Figure 13: Co-location Performance with Convolution : PIConGPU

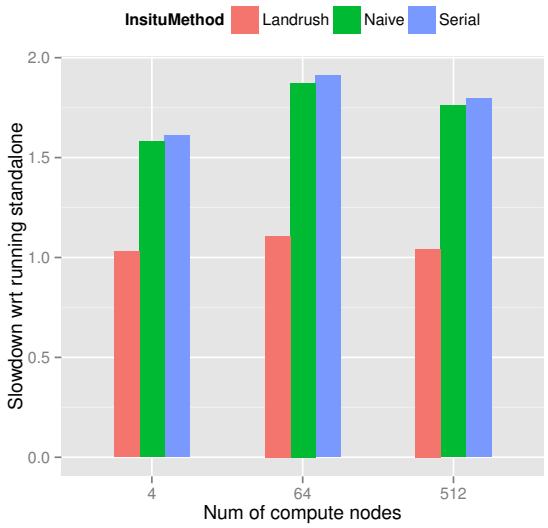

Figure 14: Co-location Performance with Convolution : LAMMPS

(as shown in Figure 12 and Figure 13) lying between 8 and 15% but Landrush is still much better keeping slowdowns under 1% for these configurations. But at 64 and 1024 nodes, serialized and naive approaches give very high slowdowns for LAMMPS lying between 76 and 91%.

Although, the slowdown for these configurations using the Landrush approach is not insignificant (4 and 11%), it is within tolerable limits to consider in situ over in-transit. This is shown in Figure 14. Overall using Landrush to run separable convolution over two dimensions, the average speedup seen is 33% compared to the best of the other approaches.

3.3.2.2 Histogram

The histogram algorithm we chose for this work performs a sparse histogram over 64 bins. An intermediate buffer is used to stage partial histograms generated by individual thread blocks running on the GPU using the shared memory to compute smaller histograms with much faster data access. Two kernels are run on the GPU with the first one generating the partial histograms and the second one merging them.

We show the performance results of running histogram in situ in Figure 15, 16 and 17. Co-location causes very high slowdowns (between 20 and 60%) at all scales for LAMMPS and PIConGPU. This is shown in Figure 15 and 16. In contrast, the worst slowdown is

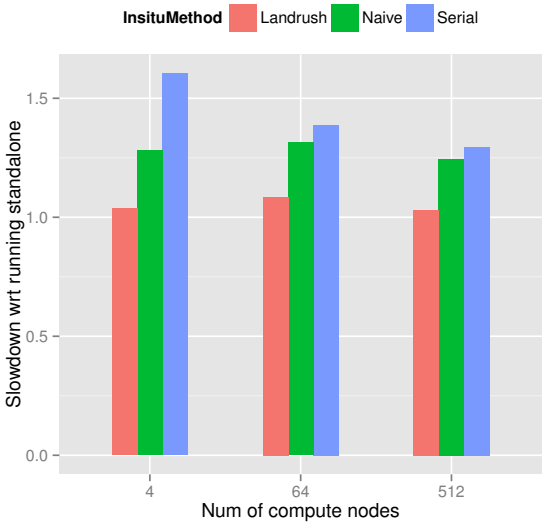

Figure 15: Co-location Performance with Histogram : LAMMPS

8.5% when using Landrush.

With GTC-P, slowdown is more tolerable using the serialized or the naive approach (3 to 14%) because timesteps in GTC-P are longer for larger runs (64 and 1024 runs), as shown in Figure 17. Overall using Landrush to run a sparse histogram in situ, the average speedup seen is 17% compared to the best of the other approaches.

3.3.2.3 Scan

Scan (or prefix-sum) is another fundamental algorithm with running time and space requirements similar to moving average computation (that is, O(N)). It is used as a subroutine in many widely used post-processing algorithms, e.g, stream compaction. We have used a divide-and-conquer version of the scan algorithm which runs three kernels on the GPU, a basic inclusive scan, an exclusive scan and a final merge step.

The performance results for co-running scan (or prefix-sum) with different scientific simulations are shown in Figure 18, 19 and 20, with the average speedup being 10%.

When the network communication delays are the smallest (that is, for 4 nodes), the Landrush approach is clearly shown to be useful giving speedups between 9 and 27%. For larger runs on 64 and 1024 nodes, LAMMPS and PIConGPU continue to benefit from using Landrush. This is shown in Figure 18 and 19.

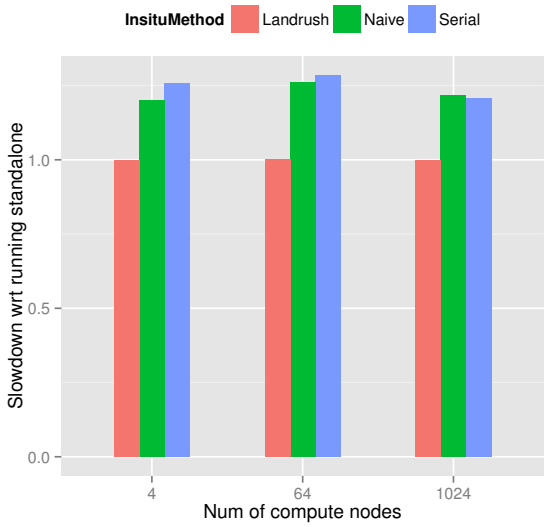

Figure 16: Co-location Performance with Histogram : PIConGPU

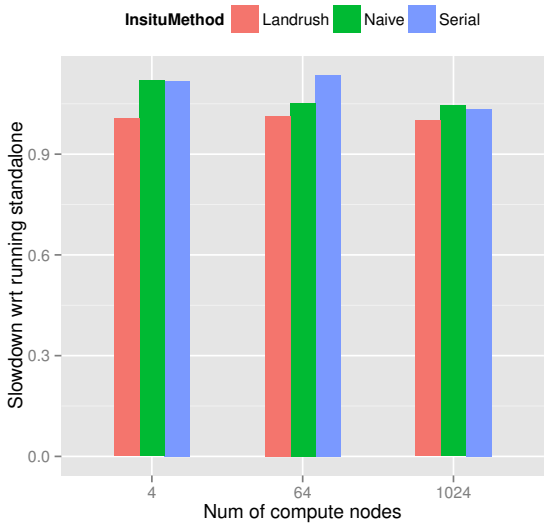

Figure 17: Co-location Performance with Histogram : GTC-P

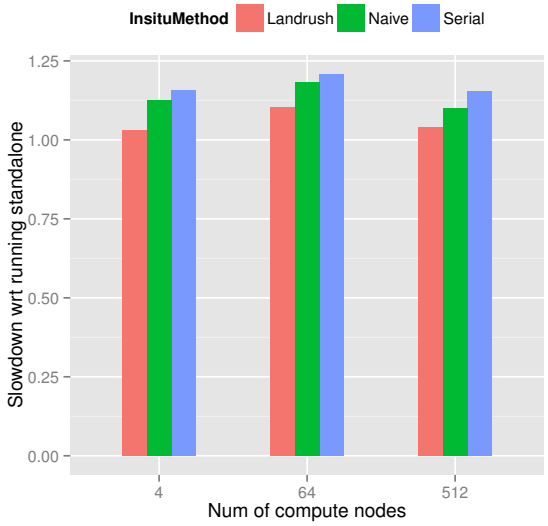

Figure 18: Co-location Performance with Scan : LAMMPS

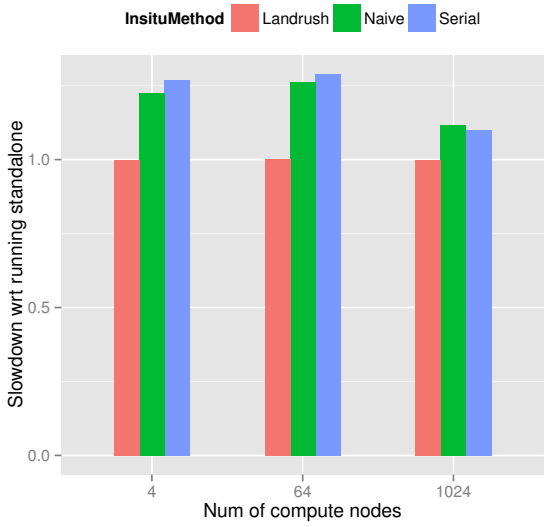

Figure 19: Co-location Performance with Scan : PIConGPU

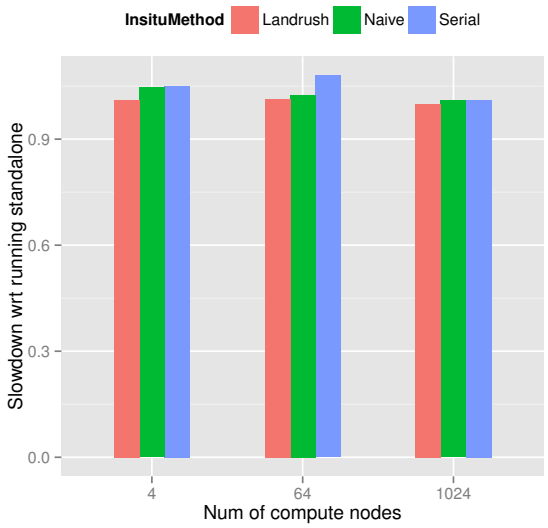

Figure 20: Co-location Performance with Scan : GTC-P

In contrast, for GTC-P, the savings diminish due to the higher running time per time step with weak scaling. This is shown in Figure 20.

3.3.3 Discussion

The average speedup obtained with using Landrush to run analysis in situ is 22, 20 and 18 % respectively at 4, 64 and 1024 nodes. This shows that longer communication episodes at higher scale-out does not diminish the importance of Landrush-style co-location of in situ analysis. Naive co-location or serialized execution is not good enough even if there is more slack. It is necessary to carefully place analysis kernels inside gaps when the science application is not using the GPU and that is what Landrush enables.

The average speedup obtained for the three scientific simulations by running analysis in situ using Landrush is 8, 19 and 33 % for GTC-P, PIConGPU and LAMMPS respectively. GTC-P scales the least linearly (running time per timestep increases for larger runs) diminishing the benefits at 64 and 1024 nodes. For PIConGPU, running analysis only inside the gaps on GPU (at most 4%, see section 3.3.1.1) is not enough. After the analysis kernels have run on the gaps on the GPU, data is moved from the GPU memory to the system memory, and limited busy-waits on the CPU limits the speedup for this application. The LAMMPS configuration used in this study has the shortest running time per timestep, but it also

exhibits the highest idleness on the GPU. The highest average speedup obtained for this application shows the importance of Landrush-style scheduling even if the GPU resource is not heavily used, like it is for LAMMPS Lennard Jones. Misplacement of analysis kernels on GPU, even if there are large gaps, can cause heavy slowdown due to the jitter caused on other application phases especially MPI communication episodes.

3.4 Chapter Summary

This work explores the implications of sharing the GPU resources on high-end machines hosting scientific simulations, to run analysis in situ. Its studies of representative high end codes demonstrate the presence of regular and usable gaps during which the GPU is unused, providing opportunities for co-running a significant amount of representative GPU-based analysis tasks on those shared resources. The positive outcome is reductions in the 'time-to-answer' seen by end users, compared to running analysis serially, when the simulation completes and/or after certain simulation time steps. Landrush, with its ability to co-locate analysis in "gaps", is shown to be superior to naive co-location that relies only on scheduling by the GPU driver.

CHAPTER IV

GPUSHARE: FAIR-SHARING OF GPU CLOUD

Following on from the scenarios in Chapter 3, GPUs are being used for more data-rich applications in cloud computing environments than ever before, and kernels in these applications have increasingly become longer in terms of running time. For example, newer machine learning kernels can run for around 100 ms vs. 637 us for the longest-running kernel across workloads evaluated in previous work([82]). Machine learning[58], graph analytics [134] and other graphics-oriented applications [90] exhibit similar long-running characteristics. However, in a cloud computing environment like Amazon's EC2, long-running kernels on GPUs can hinder scheduling flexibility and fairness for multiple-tenant scheduling because (i) GPU kernels typically run to completion and (ii) the standard mechanisms (GPU hardware schedulers and techniques like OpenCL grid offsets) don't provide the same kind of scheduling and context switching granularity as CPU threading and preemption models. This lack of scheduling granularity and flexibility for discrete GPUs drives up the cost of using GPUs in a cloud computing environment due to limited multi-tenancy and subsequent skews in fair sharing. It also doesn't allow for fair usage of GPU resources that can currently be guaranteed for CPU-based virtualized systems.

Integrated CPU/GPU systems [56] or accelerated processing units [11] have benefited from optimizations like coherent memory access with page fault handling [51] to allow for more flexibility and granularity in scheduling, but these techniques have not been fully migrated to discrete GPU environments. Previous attempts like Strings [118], disengaged scheduling [82], and elastic kernels [108] have tried to solve this problem for discrete GPU environments. However, these approaches make several assumptions that may not apply to newer, longer-running GPU kernels with high GPU resource utilization. Techniques like elastic kernels partition GPU resources such as registers and caches. However, they assume that multiple tenants are complementary, with a mix of low- and high-resource

usage kernels. Strings and disengaged scheduling use runtimes that rely on making reactive scheduling decisions based on previously completed kernels. In the case of long-running kernels, these approaches do not provide a flexible enough solution to manage kernels from different applications in a way that both maximizes GPU utilization and also provides fairness for multiple tenants.

Our approach to providing fair, granular sharing of GPUs, GPUShare, builds on previous GPU-related scheduling and middleware work and adds a "software-only yield" instrumentation to existing kernels to enable more granular context-switching in a manner that is both transparent to the user and that improves fairness between multi-tenant kernels. In addition, GPUShare differentiates between scheduling short- and long-running kernels in a way that previous work has not by providing a hybrid scheduling approach where software-based, user-level scheduling is used for managing long-running kernels and short-running kernels are handled by the existing hardware scheduler. This hybrid approach provides more opportunities for increasing fair sharing of the GPU while also ensuring the utilization loss remains low.

Our software-based middleware for fine-grained control over GPU scheduling has the following design and performance properties, which we will further elaborate in the coming sections:

- GPUShare recognizes that a distinction is necessary in scheduling short- and long-running kernels on the GPU leaving the driver to schedule the former and only intervening with scheduling actions for the latter.

- GPUShare implements and demonstrates a "software-only yield" mechanism to enable fine-grained context-switching on GPUs to enable fair-sharing over scheduling windows determined at runtime, as opposed to previous solutions that were limited in such dynamic control.

- Our experimental evaluation shows that GPUShare improves fairness in GPU use among tenants by up to 92%, incurs a maximum overhead of 12% due to context-switching in software, and scales to at least four tenants per GPU (where scalability

is limited only by the lack of memory oversubscription support in hardware).

4.1 Background

This section presents a quick refresher on how the application kernels are scheduled on the GPU by the driver and the thread management hardware. We also provide qualitative insights, wherever applicable, on how previous work on GPU scheduling fits in.

4.1.1 Programming Model, Host Application and Runtime

GPGPU programming models (CUDA and OpenCL) expose a two-level grouping of threads : (1) a thread block in CUDA (or a work group in OpenCL) is a group of threads that can communicate among each other using a software managed cache and can synchronize using barriers, in some way, equivalent to a CPU SIMD group; (2) a grid in CUDA (or an NDRange in OpenCL) is a group of thread blocks all of which run the same program (called the kernel) by a single call to the runtime. (1) provides functionality critical to any parallel programming model. It is necessary for data sharing and synchronization. Meanwhile, (2) reduces the latency of accessing the GPU by using a common framework. Grids help individual applications to maximize their GPU use but can be counterproductive to multi-tenant scheduling in the event of long runtimes. One category of previous work ([47, 113, 118, 38, 41, 121]) has proposed scheduling middleware that uses a central scheduler with "remoting" support on top of the runtime which gives good visibility of application characteristics but can introduce inter-process communication overhead if used for all kernel launches. GPUShare uses a combination of runtime profiling and prediction to delegate only long-running kernels to such a central scheduler.

4.1.2 Driver

The driver is responsible for scheduling kernels from multiple processes onto the GPU by choosing some context-switch interval. However, the scheduling actions of the driver break down if the co-located processes start issuing long-running kernels that exceed this context-switch interval. Previous work using prototype drivers([62],[82]) has tried to address this problem by controlling GPU sharing over long time intervals by measuring and

reactively scheduling the GPU use of each co-located process. This approach can be used without needing high-overhead IPC with a central scheduler, but driver-level solutions still do not provide any control on thread-block scheduling, limiting their effectiveness when long-running kernels are issued by the co-located applications.

4.1.3 Hardware Scheduling of Thread Blocks

The primary user-level control over a kernel's execution time is in choosing the size of the grid over which a kernel is launched. Once a kernel is launched with a certain grid size, the GPU hardware assigns thread blocks to Streaming Multiprocessors (SMs - multi-threaded execution units similar to CPU "cores") until all thread blocks have run. Thus, any scheduling control during the execution of a grid must reside inside the kernel and requires interaction with the running thread blocks. Previous work in the architecture space ([129],[109],[4]) has explored mechanisms to control thread block dispatch in order to enable context-switching before a grid completes execution, but this architecture support does not exist on current hardware, and it would still require additional software to accomplish higher-level scheduling objectives like fair sharing across multiple tenants. On the other hand, software[108] can be leveraged on existing hardware to run a subset of thread blocks to provide the basic mechanism for context-switching a grid. The key research challenge is determining this subset at runtime, which is what GPUShare aims to address.

4.2 Motivation

The design of the GPU driver and HW thread block management hardware are optimized to provide high GPU utilization if kernels are available to run, but this design does not address fair GPU use between two or more contexts. To motivate our middleware GPUShare, we present results from two microbenchmarks that illustrate how a tenant's share of the GPU depends on (1) the relative lengths of the running times of each tenant's kernels (shown here) and (2) if kernels are waiting to run when the driver can do a context switch.

To demonstrate point (1), we construct a microbenchmark in which the host application is able to continuously launch a test kernel of configurable running times on the GPU. It takes as input a sequence of running times and at each interval can launch a kernel with the

next specified running time. Figure 21 shows two instances of the above microbenchmark co-located on the same GPU, where each is run with a different input sequence. We chose the inputs such that the sum of the running times is same in both cases. The first four kernels issued by Process A run for 10, 10, 10 and 40 ms respectively. On the other hand, the first two kernels issued by Process B run for 50 and 20 ms respectively. Since all the kernel running times are much greater than the context switch interval of the GPU hardware (1ms), a context switch happens after each kernel execution, delaying the driver/hardware from achieving fair sharing until 140 ms. Reactive middleware, like [118], attempts to restore fair sharing earlier (for example, by allowing two kernels from Process A to execute consecutively) but still remains dependent on kernel running times. As such, they fail in this case to do any better than the driver/hardware. In order to achieve fine-grained fair sharing, time slicing of long-running kernels are necessary (as shown by the execution of the 50 ms long kernel from Process B getting sliced over four intervals) which is possible with [108]. However, in order to reduce the overhead of time slicing, some dynamic control is necessary over the time slicing interval (as shown by the last 20 ms time slice of the 50 ms kernel because a kernel from the other process, longer than 20 ms, was waiting to run). In this work, we provide mechanisms to address the challenges involved in providing such dynamic control.

Based on the above two experiments, we infer:

- Short-running kernels (i.e., shorter than the context switch interval) are not likely to affect fair GPU use and are best scheduled directly by the driver. Previous user-space scheduling middleware fails to make this distinction.

- Due to the reactive nature of the schedulers in previous work, the earliest corrective action to restore fair GPU use cannot be taken until all co-located processes have had a chance to run a kernel (in this case, after 60 ms). For long running kernels (tens or hundreds of milliseconds), this can badly interfere with achieving fair sharing of GPU across the co-located processes.

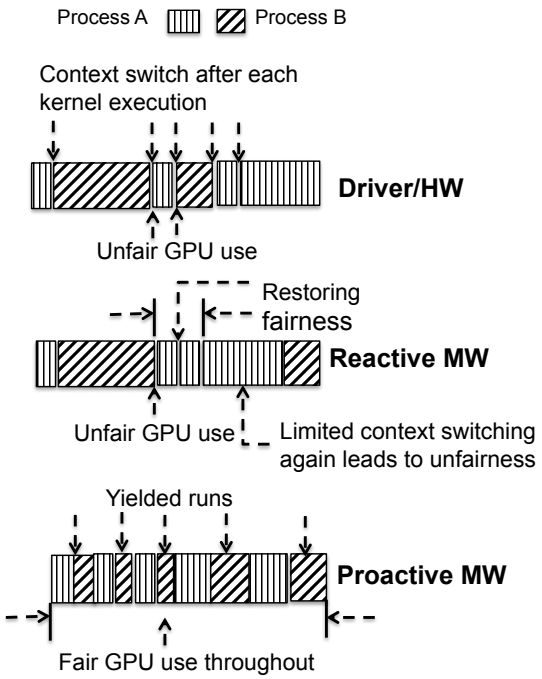

Figure 21: Granularity of time slicing with GPUShare

It is important to understand that GPUShare does not provide CPU-like context switching capability and it is still not possible to context switch at any arbitrary point during the execution of a thread block, but only at its end. Such instruction-level context switch is only possible with hardware support. In order to quantify this constraint on "yield-ing" a long-running kernel, we define its **Minimum Execution Interval** (or **MEI**) as the minimum interval of time for which the kernel runs on the GPU before it can be "yield-ed". It is equal to the interval to run a certain number of its thread blocks which is determined by the minimum of the grid size with which the kernel is launched and the number of thread blocks of the kernel that can fit on all the SMs of the GPU. If one or more of these thread blocks run for an arbitrarily long time, GPUShare cannot "yield" the kernel to allow other tenant kernels to run. In such cases, GPUShare allows more kernels from other tenants to run thereby increasing the granularity of the time window over which it provides fair sharing.

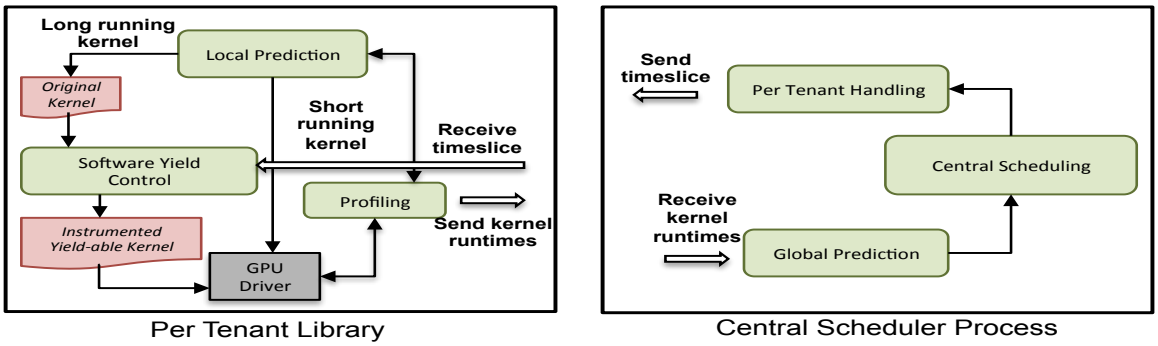

Figure 22: Interworking of different components of GPUShare

4.3 Design

GPUShare's design is shown in Figure 22. It consists of two parts: (1) a library linked with each tenant, and (2) a central scheduler process to coordinate GPU use when long-running kernels are issued. Inter-process communication is implemented using a light-weight messaging library called nanomsg [85]. Within its two part division, the different components of GPUShare broadly perform four functions as follows: (1) profiling, (2) prediction, (3) yielding and (4) global scheduling.

4.3.1 Profiling

GPUShare profiles running times of kernels on the GPU in order to predict and track running times of the same kernels when they are issued again and to distinguish short- and long-running kernels. Short-running kernels are then directed to the central scheduler for time slice allocation on the GPU. The running times of kernels are tracked for every set of parameters they are launched with and a history is maintained both locally inside each tenant process as well as inside the central scheduler. Relevant parameters for each kernel launch like the name of the kernel, its grid and thread block sizes and its shared memory use are stored in a "bin" object for each kernel.

4.3.2 Prediction

GPUShare uses profiling information to perform prediction at two places to manage scheduling of kernels on the GPU. First, the library component of GPUShare inside every tenant process predicts whether the running time of each kernel being issued is shorter or longer

than the driver's context switch interval. Second, the central scheduler predicts the remaining running time of long-running kernels waiting to use the GPU.

For the local (per-process) prediction, GPUShare uses the maximum running time from the appropriate "bin" of matching previous kernel running times. If that bin is empty, the kernel is treated as long-running, and the central scheduler is contacted. If a kernel is yielded after running for some time and reissued, it is again treated as long-running.

The central scheduler maintains some extra information to distinguish running times where all thread blocks ran to completion from those where software yielding occurred. For kernels that have not been issued and "yield-ed" before, the most recent running time from the appropriate bin is used to predict the running time. Otherwise, the running time is predicted based on the previous running time, the number of thread blocks executed in the last "yield-ed" run, the number of thread blocks remaining to complete and the allotted time slice on the GPU.

4.3.3 Yielding

When a kernel is launched with a specific grid size, all thread blocks in the grid are scheduled by the GPU hardware before control returns to software. There are two ways to slice up the total running time of the grid. (i) By sending a smaller grid consisting of a lesser number of thread blocks to run. (ii) By sending the same grid but enabling the thread blocks to finish early, if required. GPUShare uses (ii) because it gives the flexibility to choose the timeslice after a grid has been issued to the hardware. Even if all thread blocks do not take uniform time to run, (ii) is effective in controlling the desired timeslice where (i) fails.

The basic mechanism for finishing a thread block early is by executing a return statement before control reaches any of the original instructions in the kernel. The main challenge is in dynamically determining that a timeslice has expired and that the remaining thread blocks should finish early. SMs in Nvidia GPUs are clocked from different sources. With the number of SMs increasing, this may be the case with all discrete GPUs going forward. As such, the first thread block assigned to any SM sets the start timestamp for that SM. Thereafter, all the thread blocks assigned to that SM update the end timestamp for that

SM. As soon as the timeslice expiration occurs on any SM, all remaining thread blocks finish early. In this way, GPUShare is able to "yield" a grid and free up the GPU just after the end of a timeslice giving much more flexibility to the central scheduler to ensure fair GPU use.

4.3.4 Central Scheduling

The central scheduler works on the basis of (i) GPU use of both short- and long-running kernels received through profiling data from each of the co-located processes, and (ii) prediction of running times of long-running kernels issued and waiting to run. GPUShare's central scheduler takes into account per process use over the previous profiling epoch when allocating GPU timeslices to a long-running kernel where several scheduling epochs may elapse inside one profiling epoch.

For high-utilization workloads like the ones used in this work, one or more process is usually waiting to run a long-running kernel in any scheduling epoch. GPUShare limits fair-share accounting within a scheduling epoch by calculating aggregate use only at the end of a profiling epoch and then giving appropriate credit to each process for the next profiling epoch.

The GPU driver is work conserving and schedules kernels from the same process successively if others have not issued kernels in the meantime. Due to this, during each scheduling epoch, GPUShare provides a "slack window" if any process does not have a long-running kernel waiting. This guards against unfairness that may arise when some process is slightly delayed in issuing kernels, possibly due to some short-running kernel that needs to finish before it can issue the next long-running kernel. Even though the GPU use of processes with such kernel issue characteristics is nearly the same, they would fail to secure fair share of the GPU without this "slack window".

4.4 Evaluation

We ask and evaluate the following questions about the effectiveness of GPUShare:

1. Can GPUShare ensure fair GPU use between pairs of co-located tenants?

2. What is the overhead to yield a kernel?

3. Can GPUShare scale to support fair sharing for more than two co-located tenants?

Experiments were conducted on a Tesla K40 using the CUDA 7.0 toolchain and the 346.29 driver. The workloads chosen for the evaluation were nnForge[81] to run deep learning on the GPU and Gunrock [134] to run graph analytics on the GPU. Both have very high GPU utilization and represent workloads that would benefit from running on cloud-hosted GPU compute facilities. nnForge (referred to as NNF) was configured to train on two image data sets - Imagenet or IN[33] (10M images and 10K+ categories) and German Traffic Sign Recognition Benchmark or TS[50] (39K images and 43 categories). Gunrock was configured to run four analytics functions - PageRank (PR)[21], Breadth-first Search (BFS), Maximal Independent Sets (MIS) and Connected Components (CC) - also on two graph data sets - LiveJournal social network graph (referred to as SLJ)[70] with 4M+ vertices and 68M+ edges and Kronecker Graph500 logn21 (K21) [69] with 2.1M vertices and 91M edges.

The different experiment configurations for co-location studies in sections 4.4.2 and 4.4.4 are (1) default (D), where the tenant applications run unmodified and without GPUShare intervention; (2) local (L), where any tenant kernel predicted to have a running time longer than 1ms is instrumented to yield at 1ms or the kernel's *minimum execution interval* (MEI), whichever is higher; and (3) global (G), where GPU use is mediated via GPUShare with long-running kernels remoted to the backend daemon in order to enforce fair sharing. While it would be interesting to benchmark GPUShare's performance against "elastic kernels"[108] or the prototype driver used by[82], the driver used by [82] is not publicly available and "elastic kernels" requires the offline generation of a multi-threaded program for each co-location scenario where each thread is in charge of executing one of the co-located processes. For this reason, we compare against the standard CUDA runtime and hardware scheduler.

4.4.1 Metric to measure fair sharing

To measure sharing effects of GPUShare, we use a metric called *total normalized skew loss* (or TNSL) adapted from the weighted shares potential delay minimization function proposed in [80]. Instead of using the sum of weighted delays of all sharers, TNSL uses the

difference of the total number of sharers and the sum of the weighted delays. To achieve fair sharing, this value should be maximized. This is slightly more intuitive when a provider wants to distinguish users who suffered due to co-location (delay increased by more than n) from those who did not (delay increased by less than n). From a service guarantee standpoint, a provider's financial loss is directly proportional to the overly delayed users, and TNSL is intended to reflect this loss of service due to colocation. The upper bound of TNSL depends on the number of sharers - for two, three and four sharers, it is half, two-thirds and three-fourths, respectively.

4.4.2 Fair sharing between two tenants

In the first set of experiments, our goal is to evaluate the effectiveness of GPUShare in providing fair share of the GPU to two co-located tenants when they are sharing a single GPU. We pair up different combinations of workloads with varying GPU usage characteristics. But the one thing common is that all of the workloads have high utilization with hardly any gaps in GPU use.

The three workloads are - (i) neural network training using nnForge (NNF), (ii) compute PageRank (PR), and (iii) breadth-first traversal (BFS). ImageNet (IN) and German Traffic Sign Recognition (TS) are the two image databases are used for (i). A subset of the LiveJournal's social network graph (SLJ) and a network graph generated using the Kronecker product operation (K21) are used as inputs to (ii) and (iii). Fair-sharing of the GPU is measured using TNSL and results are shown in Figures 23, 24 and 25. The co-location scenarios evaluated are - NNF-IN and NNF-TS with PR-SLJ (Fig. 23); NNF-IN and NNF-TS with BFS-SLJ (Fig. 24); PR-SLJ and BFS-SLJ; and PR-K21 and BFS-K21 (Fig. 25).

The salient characteristics that affect sharing depend on both the workload and the dataset. For example, TS is a smaller image database than IN the result of which means that NNF does not issue any long-running kernels (>1 ms) when run using the TS data set. As such, local-only yielding (config L) of co-located kernels is not enough to ensure fair-sharing. Among the graph algorithms in the Gunrock library, BFS is a single traversal algorithm with variability in its long-running kernels' execution time. It has only one kernel

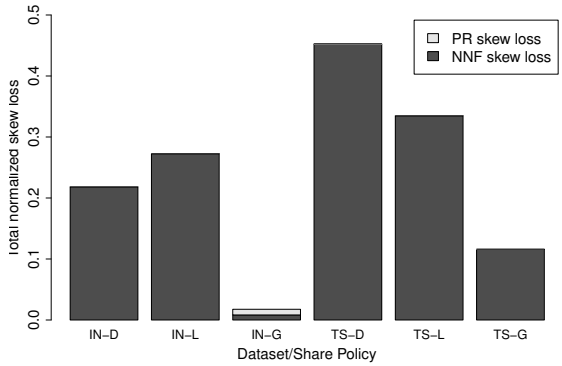

Figure 23: NNForge and PageRank

to be issued, which is not amenable to yielding because of a high MEI of 12 ms. In contrast, PR is an iterative algorithm involving repeated issuing of long-running kernels and the central scheduler in GPUShare is needed to enforce fair sharing (config G).

Config G is the most effective in all of the co-location scenarios to reduce TNSL ranging from at least 68% (BFS and NNF-TS in Fig. 25) and going up to 92% (PR and NNF-IN in Fig. 24) better than scheduling done by the driver and the hardware scheduler (config D). If there are not many long-running kernels with MEI higher than 1ms, as is the case with BFS and NNF (Fig. 24), fair sharing can be achieved just by yielding each kernel at 1 ms without using the central scheduler (config G).

In summary, the local-only yielding of long-running kernels after 1 ms or the MEI is not enough to ensure fair sharing whenever there are differences in MEI or number of long-running kernels issued by tenants. In such cases, GPUShare's central scheduler, which is equipped with global profiling information, is necessary.

4.4.3 Overhead

Here we evaluate the overhead for using the GPUShare middleware versus the standard NVIDIA hardware scheduler.

First, we look at the overhead incurred to yield a kernel at the boundary of a thread block based on the expiration of an allotted time slice or yield interval. Figure 26 shows the slowdown of NNF-IN, BFS-SLJ, and PR-SLJ for different yield intervals from 1ms up to 128ms, which is longer than any of the kernel runtimes. The highest overhead is observed

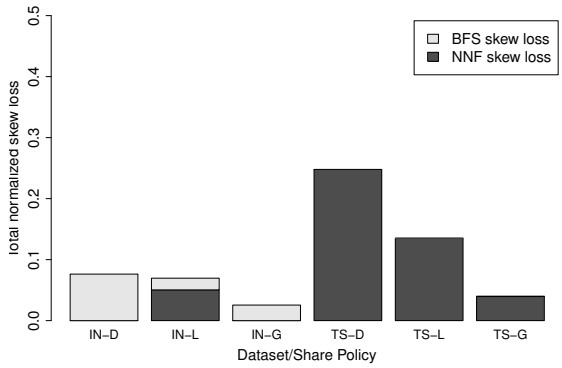

Figure 24: NNForge and BFS

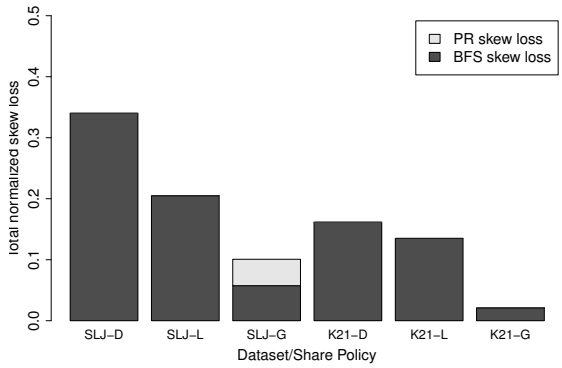

Figure 25: BFS and PageRank

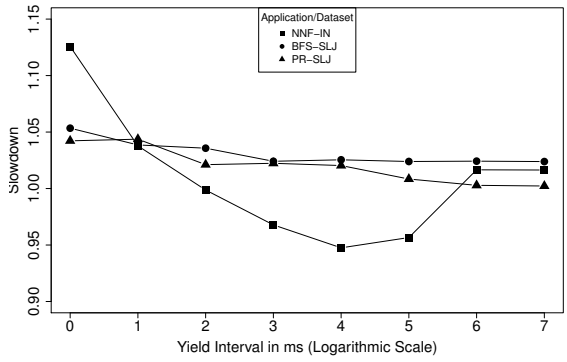

Figure 26: Local Only Yield : Slowdown at different yield intervals

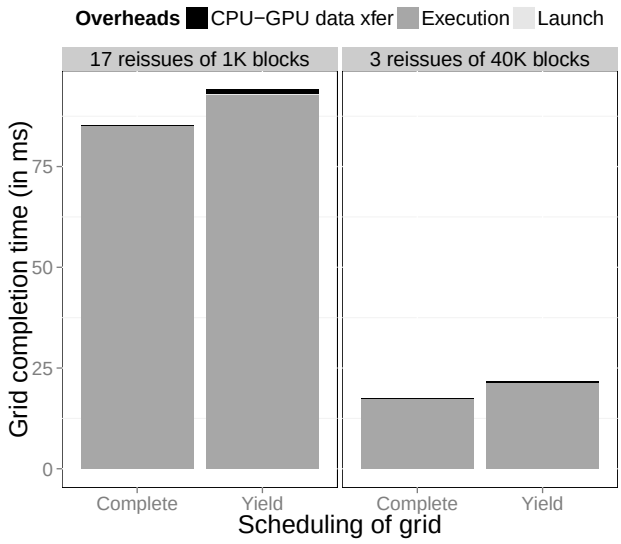

Figure 27: Breakdown of yield overhead

for the smallest yield interval of 1ms, which causes the maximum amount overhead for BFS-SLJ and PR-SLJ at around 4% and 12% for NNF-IN. However, NNF-IN overhead drops at the 2ms yield interval and actually becomes negative at 4ms-32ms due to the effect of limiting the number of concurrently running thread blocks.

We further break down the total overhead in an attempt to segregate it into two parts: 1) the cost to reissue a yield-ed grid multiple times and 2) the cost of running the grid in a time sliced fashion. We select two kernels launched with very different grid sizes and number of reissues needed to complete the respective grids with the yield interval set to 1ms. The "advance" kernel in the "pagerank" workload is launched with a grid size of only 1024 thread blocks and has to be reissued 17 times, while the "filter" kernel in the "connected components" workload is launched with a grid of nearly 40K thread blocks and gets reissued only 3 times. As Figure 27 shows, the overhead due to (1) launch (ie yield and reissue) and (2) data transfer for time slice and progress data is relatively small when compared to the (3) execution time of kernel, and larger grids (ie, the filter kernel in CC) do not show longer execution overhead. Both of these results demonstrate that the yield overhead is reasonably small. Inspection of these results have also shown that execution time overhead is mostly tied to lost cache locality for reissues.

Next, we look at the combined effect of yield intervals and the latency of communicating

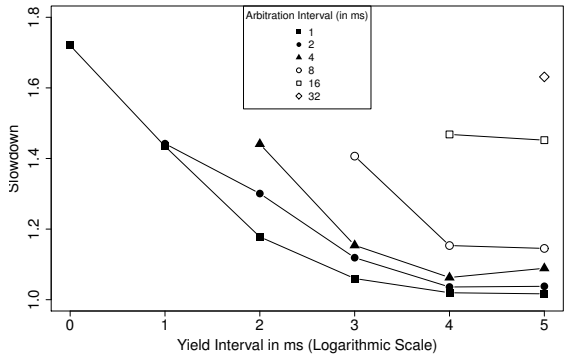

Figure 28: Neural Net Training on Imagenet : Slowdown at different yield intervals and arbitration epochs

with a central scheduler and how these factors affect a tenant application's overall running time. The number of messages between the global daemon and a tenant is controlled by the choice of yield interval for a kernel. In this set of experiments, we vary the yield intervals starting from 1ms going up to 32ms, in powers of 2 while the central scheduler's arbitration epoch varies from 1ms up to the length of the yield interval.

Figure 28 summarizes the results for NNF-IN. When the arbitration epoch and the yield interval are both short (1ms), the overhead is very high (1.7x slowdown) due to the large number of messages between the central scheduler and the tenant process. But for short arbitration epochs (1ms), messages can be reduced by increasing the yield interval so that the application is blocked less often. For example, an 8ms yield interval only has 5.6% overhead.

On the other hand, arbitration epochs any longer than 4ms need very high yield intervals, which in turn affects fairness, as shown in the high performance sensitivity for PR-SLJ with a yield interval of 16 ms (Figure 29). This slowdown is due to the unequal decrease in number of skipped thread blocks going from an yield interval of 8 ms to 16 ms compared to going from 4 ms to 8 ms. BFS-SLJ (results in Figure 30) issues a smaller number of long-running kernels than NNF-IN and PR-SLJ over the same interval of time and thus has less communication with the central scheduler and flat overhead for varied yield intervals.

To summarize, a yield interval of 4 to 8 ms is necessary for tenants that issue many long-running kernels in order to limit the number of messages to/from the central scheduler.

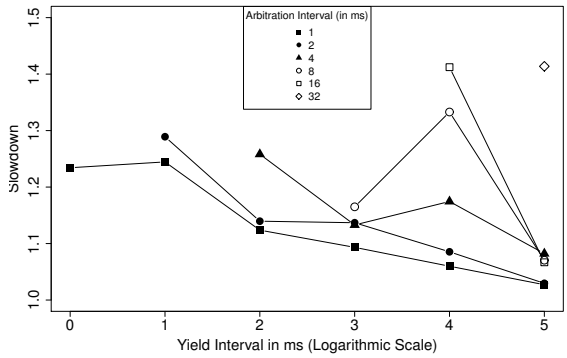

Figure 29: Pagerank on SocLiveJournal : Slowdown at different yield intervals and arbitration epochs

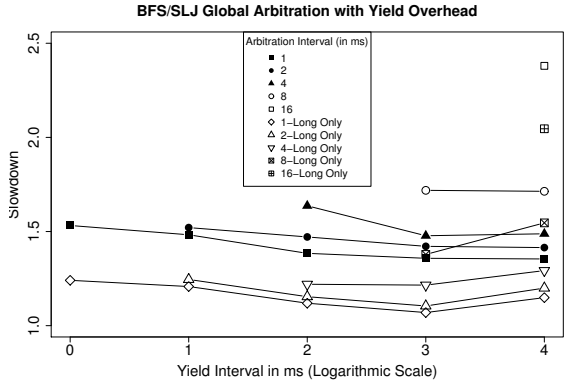

Figure 30: Breadth-first-search on SocLiveJournal : Slowdown at different yield intervals and arbitration epochs

This avoids slowing down tenant processes too much. It is also observed that the central scheduler's arbitration epoch can't be more than the selected yield interval if latency of response from the scheduler needs to be overlapped.

4.4.4 Scalability

To test the scalability of our middleware, we study the effect of co-locating three and four tenants together on the same GPU. The main constraint to increasing the number of co-located tenants further is the amount of on-board device memory and the current lack of support for paging of GPU memory.

The worst-case TNSL for any of three tenants is 0.66 while it is 0.75 for four tenants. While not worst-case, Figure 31 demonstrates that many configurations with the default scheduler suffer from high TNSL (all cases except BCM-D where BFS, CC, and MIS are colocated by the default hardware scheduler). This is because PR (PageRank of LiveJournal graph) is one of the tenants in each of the remaining cases. Due to PR's continuous supply of long-running kernels, it always ends up with greater share of the GPU at the expense of the other tenants.

The local-only yield policy (config L) is very effective at improving TNSL when fair sharing is impacted due to long-running kernels with low MEI (such as in PR). By each tenant yielding locally, the wait times of other tenants that were previously blocked by the long-running kernels of PR are reduced and the TNSL improves between 32% and 89%. The only case where this doesn't work is for BCM-L because each of BFS, CC and MIS have long-running kernels with high MEIs which differ.

Using GPUShare's central scheduler to do global arbitration (config G) helps to improve TNSL by 23% to 78% over local-only yield and by 61% to 76% over the default hardware scheduler. Global knowledge of MEIs helps to compensate for the difference in MEIs of kernels issued by BFS, CC and MIS which fixes the TNSL for BCM-G. Central scheduling fails to improve TNSL when kernels with lower MEIs have longer running times because it adds extra overhead to such tenants for sending/receiving messages. This is why the TNSL deteriorates for PCM-G (PR, CC and MIS) compared to PCM-L.

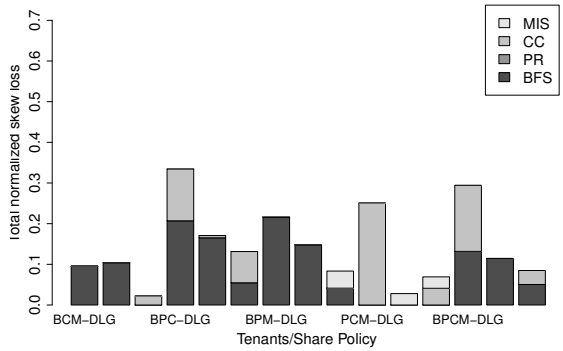

Figure 31: More than two Tenants (3 and 4)

4.5 Chapter Summary

The present approach to GPU multitasking defers all context switches to kernel boundaries, but this technique does not enable fair sharing and SLO guarantees for big-data applications running in multi-tenant datacenter environments, which increases the costs of using GPUs in a cloud environment. GPUShare improves on the state-of-the-art by offering fine-grained context-switch functionality (at thread-block boundaries instead of kernel boundaries) via a low-overhead instrumentation of GPU kernels, a middleware wrapper layer, and a software-emulated yield of processing cores. It also offers a hybrid scheduling method for long- and short-running kernels to reduce overhead.

CHAPTER V

SYMPHONY : A SOFTWARE-SUPERVISED, MULTI-KERNEL SCHEDULER FOR IN SITU GPU WORKFLOWS

High-performance computing (HPC) machines that use GPUs for compute acceleration are currently burdened with the challenge of balancing bandwidth needs between the CPU, GPU and the I/O systems. This limited bandwidth motivates the use of on-GPU in situ analysis for scientific simulations to reduce data movement over PCI Express (PCIe) and to improve the overall performance of combined simulation and analysis runs for scientific applications[42]. While in situ analysis provides significant advantages in reducing time to completion, it requires the ability to co-schedule compute kernels from multiple applications on the same GPU.

For the current generation of GPUs, scheduling of thread blocks inside compute kernels is done by a hardware scheduler, which minimizes scheduling overhead but also limits the flexibility of the generated schedule. Current GPUs focus on scheduling kernels so that they complete in the shortest amount of time, but the GPU hardware scheduler's policy might hurt the combined progress of a co-running application by requiring added data movement when switching between simulation and in situ analysis kernels. In addition, current GPU hardware schedulers typically can not predict opportunities to interleave in situ kernels with simulation kernels or to reuse existing data on the GPU for in situ analysis. These limitations prevent effective usage of the GPU for in situ analysis.

The majority of previous work focusing on GPU multi-tenancy of compute kernels has looked into data center-like sharing environments where per-requester performance instead of overall workflow performance characteristics was considered ([118], [82], [132], [127]). There has been more recent work that has considered HPC workflows ([145]) but even in such work, the hardware scheduler in the GPU limits the flexibility of such mechanisms.

To address the problem of effecting co-scheduling simulation and analysis on GPUs, we

have developed a system called *Symphony*, a software-supervised, multi-kernel, thread block scheduler for GPUs, and evaluated it. Specifically, the contributions of this work are the following-

- We propose a *GPU-resident* software scheduler to schedule thread blocks of compute kernels from multiple applications, specifically to improve time to answer for scientific applications and related in situ analytics. To the best of our knowledge, this is the first work to propose GPU-resident scheduling of threads as opposed to host-based software scheduling. These resident threads are similar to operating system threads running on CPU cores in that they can be used to run application code on demand.

- Our evaluation of this approach demonstrates that different application and analytics combinations have different amounts of "headroom", and we use these experiments to reason about the scenarios when in situ analysis is most beneficial, how much resources are needed to schedule analysis on the GPU in relation to a given scientific simulation, and what types of analysis match best with the given specific application.

- Finally, our approach shows that the time to solution is improved by *Symphony* in the range of 15% to 30% for three common scientific codes and relevant analytics routines. Overhead of scheduling in software is reasonably low ranging from a speedup for one of the scientific codes (of 8%) to a slowdown of 18%. We also show that overheads can be mitigated by using batching to reduce the frequency of software intervention.

This work focuses on NVIDIA GPU hardware (as used in systems like ORNL's Titan[105] and Summit[106] supercomputers), but the proposed technique applies equally well to other discrete GPUs.

5.1 Motivation for GPU-resident Coscheduling of Analytics

GPUs currently schedule computation at the granularity of kernels, and when a scientific application launches a GPU kernel, the GPU driver and hardware-based scheduler allocate all of the device's cores (symmetric multiprocessors or SMs in Nvidia terminology) to the kernel. Unlike CPU schedulers, GPUs are designed with the goal of keeping the device

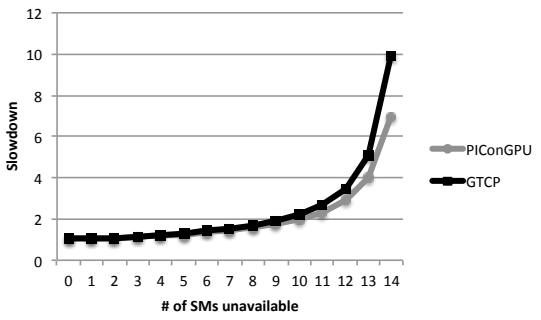

Figure 32: Impact of available SMs on throughput of two well-known PIC codes

busy with highly parallel tasks in order to achieve best performance on a single kernel and high utilization by eliminating software intervention during the kernel's execution. For this reason, GPU applications typically launch a much larger number of SIMD groups of threads or a "batch" of thread blocks (a SIMD group of threads in Nvidia terminology) than the available physical cores on the GPU in order to amortize the latency of launching a kernel. One result of this batch-based model of execution of kernels on GPUs, is the inevitable head-of-the-line blocking that occurs when a very large batch (a "grid" in Nvidia terminology) is scheduled to run and enjoys exclusive access to the GPU for a relatively long period of time. This is quite common for GPU kernels in scientific simulations that operate on large amounts of data resulting in big "grids".

This model of granting exclusive access to a single kernel severely affects the amount of time available to do useful in situ analytics, particularly when the co-running science code is characterized by many such large "grids" that have very high utilization of the GPU. Ideally, we would like to allow for a scheduler that can make scheduling decisions "during" a GPU kernel's execution in addition to scheduling points only "after" the execution of the kernel that is possible today.

We perform a characterization of two particle-in-cell (PIC) plasma simulations, PICon-GPU [22] and GTC-P[137], both of which use the GPU for computing the most compute-intensive kernel, current deposition. During the current deposition phase, particle (ions and electrons) properties remain static or consistent and are good candidates for in situ analysis. Our characterization looks at the effect of available HW parallelism on the throughput of

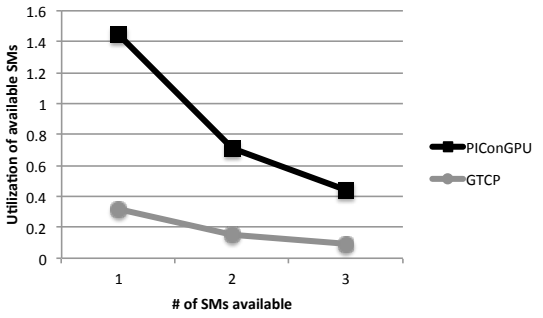

Figure 33: Utilization of available SMs for histogram analysis

the two scientific applications, which can be indirectly controlled by limiting the number of SMs that kernels from PIConGPU or GTC-P run on the GPU. The resultant slowdown versus number of SMs allocated per application is shown in Fig 32. Although these two simulation kernels differ greatly in implementation, they exhibit strikingly similar scaling characteristics with a sublinear slowdown for the first few SMs. In other words, the fractional performance loss of the simulation kernels running on a few SMs less than normal is quite reasonable for both tested applications. With up to 3 SMs limited or "revoked", the slowdown observed is within 15% of the "hardware-scheduled" application running on all SMs.

We also characterize the performance of a GPU histogram analysis application that runs on the particle data of PIConGPU and GTC-P using up to three SMs. Here we assume that these SMs have been revoked from the respective simulations and allocated to the histogram application's kernels. We measure the running time of histogram normalized to the charge deposition time intervals in the related simulation code. The rationale is that the charge deposition interval is the maximum time available for the histogram analysis operation to complete while the particle data remains consistent. If the normalized running time is greater than one, it implies that the simulation needs to be stalled for analysis to finish after the charge deposition has completed. Fig. 33 shows that for both the simulations, it is possible to run the histogram analysis on the GPU simultaneously with the charge deposition phase without causing any stall and by using fewer than three SMs. These simple experiments demonstrate that even the parallel efficiency of optimized HPC applications

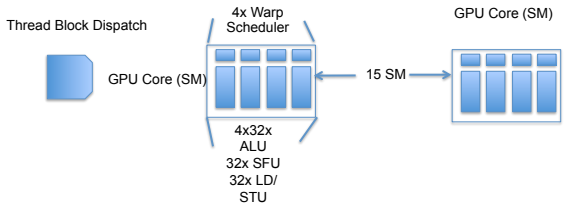

Figure 34: Processing cores and scheduling hardware in modern GPUs

cannot fully leverage the large number of SMs in GPUs on high-end machines. As a result, there is some slack available to schedule in situ analysis kernels on the GPU that can run during windows when the data to be analyzed remains read-only.

5.2 Design

Fig 34 shows the HW components of a GPU core and the scheduling function present inside most common GPUs today. In terms of ALU hardware, symmetric multiprocessors (or SMs in Nvidia terminology) are similar to SSE-enabled CPU cores with hyper-threading. The key difference arises in the way a SIMD thread's state is managed during its lifetime. The GPU manages it entirely in hardware using the warp scheduler whereas the operating system/thread library is in charge for the CPU. Thread stop/resume/migrate decisions on the CPU occur due to I/O requests from executing threads and/or OS scheduling events. CPU threads in a system do not often increase in excess of available processing cores, but an SM on the GPU mostly operates in an oversubscribed state with upto sixteen times more SIMD threads than available cores. The warp scheduler's job is to quickly switch out SIMD threads waiting on long-latency load/store requests with others that have ALU operations to run. The thread block scheduler operates at a level higher where it balances load across the SMs feeding SIMD threads into each of the warp schedulers.

It is important to observe that all these HW resources on the GPU are available to a single kernel launched from a single application context for the duration of execution of the kernel. For single-application usage scenarios on GPU, the hardware-managed approach works perfectly. However, to support fine-grained sharing of GPU cores across applications for in situ analysis, SMs need to be independently schedulable entities. We intend to achieve this with *Symphony*, a software layer on the GPU similar to the OS scheduler/threading

library that mediates execution of different application kernels. Presented below are some of the design considerations in implementing such a system.

5.2.1 Warp-level vs Thread block-level abstraction

Abstracting scheduling control at the level of warps can provide great flexibility but may turn out to be counter-productive due to the extremely low-level monitoring information needed to schedule efficiently. For example, heavy instrumentation is required to identify long latency load and store requests. *Symphony*, thus, focuses on abstracting scheduling control at the granularity of thread blocks of application kernels. To eliminate the effects of destructive interference as far as possible, we have chosen to use a homogeneous distribution of thread blocks among the available SMs where any given SM only runs thread blocks on a given SM are from the same application kernel.

5.2.2 Hardware thread block scheduling - ignored or software supervised?

Today, the hardware thread block scheduler in GPUs is not a directly programmable unit that can be reconfigured during the execution of an application kernel. For the sake of argument, even if that is implemented in the future, we believe that a software entity on the GPU-side needs to exist to manage the reconfiguration to best adapt to co-running applications' performance goals. The device driver approach of the host operating system will simply not scale to support such fine-grained control without incurring a huge amount of overhead. In *Symphony*, we have used the software supervisor approach where a supervisor kernel cooperates with the hardware thread block scheduler to schedule application kernels on the SMs. Due to current hardware limitations, the hardware thread block scheduler's role is reduced to starting the supervisor kernel(s). Thereafter, the rest of the scheduling responsibility is undertaken by the supervisor kernel without any assistance from the hardware thread block scheduler. However, it is possible that a future version of the supervisor kernel could delegate scheduling responsibility to a compatible hardware thread block dispatcher momentarily to avoid the overhead of software control.

5.2.3 Dynamic vs static resource allocation

As previous work[43] has shown, it is possible to allocate kernels to the GPU in conservative steps that are known statically. By launching no more thread blocks than the occupancy of a given kernel on a particular GPU, it is possible to ensure much better fairness in sharing the GPU across multiple application kernels. However, this approach does not provide any response time guarantees because the issued grid of thread blocks across all SMs may take an arbitrarily long time to release the GPU. Even if there are some thread blocks that have completed execution, those SMs do not become available for use by other application kernels. *Symphony* is designed so that it can utilize such thread block idling because it manages application kernels at the granularity of individual thread blocks instead of complete grids.

5.2.4 Additional offload APIs vs more compiler directives

Some of the basic interactions between an application and *Symphony* have to rely on the offload API provided by the GPU vendor (for example, CUDA in case of Nvidia GPUs). For example, bootstrapping the supervisor kernel or getting application kernels through to the supervisor kernel, requires the device driver's intervention. But more fine-grained scheduling hints are better inserted in the form of compiler directives similar to existing standards like pragmas with OpenACC[107]. For example, *Symphony* supervisor kernels are shape-preserving with respect to the shape of the thread blocks of the application kernels they run. Scheduling hints could be used to assist libraries like *Symphony* in supporting multiple supervisor kernels with compatible shapes of the application kernels that have been communicated via device code or compiler directives. In addition, some or all thread blocks in a given application kernel might be marked as shape invariant such that they could be executed by different supervisor kernels, which would offer greater scheduling flexibility. While the current version of *Symphony* does not support these compiler directives yet, this is a future research area for improving the co-location of different kernels.

5.3 Implementation

Symphony adds software supervision to the hardware-only thread block scheduler of GPUs by delegating the hardware to schedule only a small number of thread blocks and using software to schedule the rest. For large scientific applications and their associated analytics operations, most of the GPU running time is composed of a few long-running kernels. Thus, performance can be influenced to a large extent by the software scheduling decisions and the hardware scheduler can be leveraged, when necessary, to minimize the overhead. *Symphony* consists of three components - (1) the *deployer* is responsible for deciding resource partitioning between science and analysis kernels and setting up software supervision for their GPU execution; (2) the *orchestrator* performs the actual supervised scheduling of thread blocks once simulation and analysis kernels have been deployed; and (3) the *consolidator* takes care of reassigning SMs as they become available when one of the kernels completes execution and the deploy-time allocation becomes stale.

5.3.1 Deployer

The deployer block resides entirely in the application's host-side user space. *Symphony* implements a wrapper layer around the CUDA runtime/driver to intercept application kernel launches to the GPU. It uses a simple heuristic based on the CUDA occupancy calculator API to determine if the issued kernel will be long-running and software supervision is necessary. If the size of the grid is 10 times more than the occupancy, *Symphony* considers the kernel to be long-running and deploys a *supervisor kernel* by wrapping the original application kernel with *Symphony* scheduling code. Any long-running kernel needs to be JIT-compiled just once to make it *Symphony* aware, and then it can be cached for subsequent use. *Symphony* also uses the CUPTI profiling API to continuously monitor GPU kernel execution times of the scientific simulation and the analysis application, so it can identify other long-running kernels with smaller grids that were previously not considered by *Symphony's* heuristic for supervised scheduling. *Symphony* follows a conservative approach for co-scheduling analysis by using software supervision for all kernels except ones with a grid size less than or equal to the occupancy determined by the occupancy calculator. The

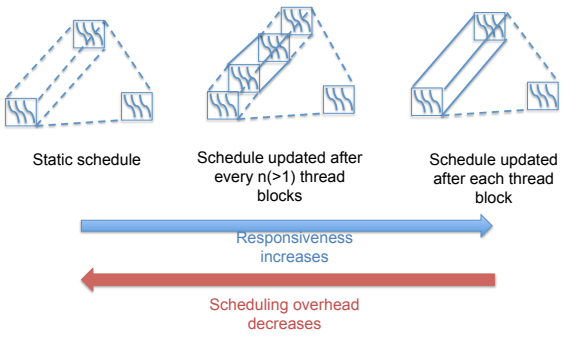

Figure 35: Software-supervised thread block scheduling : Responsiveness vs Overhead

module loading API in the CUDA driver wrapper of *Symphony* can also be used to point to only the kernels that a given user wants to be scheduled with software supervision.

Once an application kernel is identified by *Symphony* to be scheduled with software supervision, a shape-preserving supervisor kernel is launched with identically shaped thread blocks and a grid size equal to the occupancy of the kernel determined by the occupancy calculator API. The supervisor kernel runs the first N thread blocks of the application kernel where N is the occupancy. *Symphony* follows a 1:1 mapping where each supervisor kernel thread block executes a single application kernel thread block. This is followed by launching progressively smaller supervisor kernels of grid sizes N(M-1)/M, N(M-2)/M... etc. that are allowed to run on (M-1)/M, (M-2)/M... etc where M is the total number of available SMs. In other words, a linear search is performed to determine the number of SMs that yields the best value of resource scale down (number of SMs used/total number of SMs) to slowdown (running time using the allotted SMs/running time using all the SMs). *Symphony* stops the linear search as soon as the resource to slowdown ratio decreases or the number of SMs allocated to the scientific application is half of the total SMs. For the analysis kernels, the search is carried out in the reverse direction, and the search is stopped once the speedup to resource scale up ratio decreases or the number of SMs allocated is equal to the leftover SMs from the simulation kernel's search. Once a supervisor kernel is deployed, the orchestrator embedded into the corresponding *Symphony*-aware application kernel gets notified of the number of completed thread blocks that it took to determine the SM allocation.

5.3.2 Orchestrater

The orchestrator resides on the GPU-side and consists of two components. The first component is a kernel-embedded fixup code for grid size and thread block identifier, which preserves the application's semantics wherever any assumptions about grid size and thread block identifier are made. This prevents existing applications from breaking when application-specific grids are altered and resulting thread blocks are scheduled by *Symphony*. The second component is the scheduling logic that resides inside each supervisor kernel for executing the science kernel and the co-scheduled analysis kernel(s).

The main challenge in the implementation of the fixup code is the degree of pointer aliasing used in the application kernel. For the very few kernels with pointer aliasing (none in the HPC applications tested here), we manually instrument the kernel source for *Symphony* support. We believe pointer aliasing of grid size and thread block identifier in kernel sources are avoidable without performance penalty such that the compiler can enforce needed checks to prevent issues with the fixup code.

Symphony provides scheduling flexibility through supervisor kernels which can schedule thread blocks using a wide range of scheduling policies. In this work, we have focused on the frequency of software intervention and how it can be used to trade off scheduling overhead against combined simulation/analysis throughput. However, a supervisor kernel can be imagined as a distributed scheduler consisting of multiple thread blocks running on one or more SMs where each supervisor kernel thread block runs some number of thread blocks of the application kernel. The scheduling questions we attempt to answer are - (1) what is the mapping of application thread blocks to supervisor kernel thread blocks?, and (2) how is the mapping arrived upon?

(1) involves exploration of a huge index space for even moderately sized grids because the GPU programming model does not impose any restriction on the order in which thread blocks of a grid are scheduled. However, data locality characteristics between thread blocks and nearby indices for most application kernels motivates us to start with a strided assignment of application thread blocks across SMs. For (2), a static mapping might lead to better performance than the hardware-only thread block scheduler because a few supervisor

kernel thread blocks have to be scheduled instead of many application kernel thread blocks. However, there is a potential load imbalance between supervisor kernel threads where a static quota of blocks might result in idle cores. A runtime decision where each supervisor kernel thread block contends for the thread block to run can make the scheduling policy highly responsive to load imbalance, but serialization for a supervisor to decide on the next thread block can inflate overhead. This trade off is shown in Fig.35. *Symphony* take the middle ground by supporting a batch assignment where a supervisor kernel thread block can choose N application kernel thread blocks to run at each contention point. N can be large when the original application kernel grid size is also large and vice versa.

5.3.3 Consolidater

The consolidater component of *Symphony* exists on both the host and the GPU. On the host, the consolidater's task is to launch a scaled up supervisor kernel as a result of SMs becoming available due to the simulation or the analysis kernel completing early. We intentionally implemented the consolidater as separate from the deployer. In principle, it should be possible to launch additional supervisor kernels from the GPU using CUDA dynamic parallelism[94] or OpenCL device-side enqueue[64]. Future GPUs might expose SM allocation control to software making multi-kernel SM sharing easier to manage. If so, the consolidater could exist entirely on the GPU.

There are two situations that could arise once simulation and analysis supervisor kernels have been deployed. In one scenario, the analysis kernel may finish execution early in which case SMs become available for the simulation kernel to use. In the other scenario, the simulation kernel may be done while the analysis kernel is still running and subsequent simulation kernels have to be scheduled. When there is data consistency issue by letting other simulation kernels run, the analysis kernel should scale up to use the available SMs in order to complete quickly. However, when that is not the case, a paritioning decision is necessary to allocate the SMs that became available. *Symphony* does not recalculate SM distribution but simply allocates the freed SMs to the next simulation kernel. For all cases where the remaining time to run the analysis kernel is less, the overhead of recalculation

can be avoided.

The supervisor kernel to be scaled up is stopped by mimicking a software mailbox. Whichever supervisor completes first posts to the mailbox. The mailbox is checked by each thread block of a supervisor (in this case, the one that is still running) during batch assignment. So, large batches can cause some delay in effecting the consolidation. Thereafter, a new supervisor kernel is launched which runs on all available SMs. State is shared between the consolidater and the orchestarter to checkpoint where in the grid did the consolidation occur. Also, the fixup code gets updated information necessary to guarantee launch-dependent application semnatics.

5.3.4 Limitations of the *Symphony* Approach

Symphony allocates computation resources to supervisor kernels at the granularity of SMs, but *warps* in Nvidia GPUs and *wavefronts* in AMD GPUs are actually the lowest granularity at which hardware resources can be assigned to software. When the kernel's thread block size is larger than the size of a warp, allocation at the granularity of a *warp* presents multiple performance challenges in the current hardware. In particular, the faster access *shared memory* cannot be used for most access patterns, and software barriers are required for *syncthreads*, which would otherwise be handled in hardware. For this reason, *Symphony* currently schedules only at SM-sized granularities.

Also, kernel sources need to be available in order to schedule them using *Symphony* due to the need to slightly modify launch parameters to work with a superkernel launch. Although most HPC codes are open source, some important analysis kernels like FFT routines in CUDA [101] are distributed as binaries. However, it should be noted that performant open-source libraries are also available([136], [55]) which could be used to leverage the advantages of *Symphony*.

5.4 Evaluation

In evaluating *Symphony*, we ask the following questions:

1. Is it able to improve the overall *throughput* of scientific and analytic workflows?

2. How much is the *overhead* compared to the HW scheduler when in situ analysis on the GPU is not used?

5.4.1 Setup

To evaluate *Symphony*, we look at three common scientific applications (PIConGPU, GTC-P, and LAMMPS) and related analysis tasks (Histogram, Nearest-Neighbor) that can be run in situ on a GPU with the co-located scientific application using *Symphony* to manage scheduling. While some single node testing and debugging is run using an NVIDIA K40 GPU, these application and analytics routines are primarily tested using Titan with 64-node jobs where each node has an AMD CPU and an NVIDIA K20 GPU spanning over 1024 CPU cores and 793 symmetric multiprocessors. We have used a high-performance GPU library, called ArrayFire [140], to build the analysis workloads that are co-scheduled with scientific simulations.

PIConGPU is a particle-in-cell (PIC) plasma simulation where all computation runs on the GPU. A small number of CPU cores (1 out of 16 on Titan) are used for launching kernels on the GPU, moving data to/from the GPU memory and exchanging data with MPI processes. This limited use of CPU cores leaves them available for analysis, but limited PCIe bandwidth becomes a bottleneck when the simulation is run with problem sizes that come close to filling GPU memory.

GTC-P is similar to PIConGPU in that it is also a particle-in-cell plasma simulation, and the data consistency property of the particle data during the charge deposition phase can be similarly exploited to run analysis on the particle data. However, it differs greatly from PIConGPU in its implementation with a mix of parallel compute running on the CPU cores and the GPU. Similar to PIConGPU, the charge deposition phase of GTC-P also runs off the GPU and constitutes around 65% of the running time in any given time step. In the context of this work, we investigate the effects of co-running analysis kernels on the GPU.

LAMMPS or Large Scale Atomic/Molecular Massively Parallel Simulator represents another important class of scientific simulation, namely, molecular dynamics. We configure LAMMPS to simulate Lennard Jones potential using a cutoff distance. This configuration

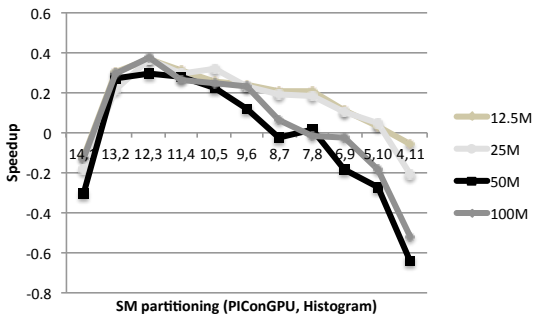

Figure 36: SM scaled throughput of PIConGPU and Histogram workflow

has multiple applications, such as simulating cracking and melting of materials. We use the USER-CUDA package in LAMMPS (as opposed to the LAMMPS GPU package) due to its higher GPU utilization, which makes co-location of analysis kernels on the GPU more challenging.

5.4.2 Throughput

In the first set of experiments, we test the effectiveness of the space sharing capabilities of *Symphony* in providing improved throughput for different scientific workflows exhibiting diverse GPU execution characteristics.

5.4.2.1 PIConGPU and Histogram

For PIConGPU, the particle data constitutes the bulk of the memory requirement of this simulation, and histogram is a frequently used analysis performed on various particle attributes like particle energy etc. A time step in PIConGPU consists of charge deposition which constitutes 70% of the running time. During this interval, the particle data is not updated. This implicitly provides data consistency so that read-only analysis can be run. If the histogram calculation does not complete inside the time it takes for charge deposition to finish, PIConGPU is stalled. Alternatively, *Symphony* also enables the analysis calculation to be halted and the results to be discarded.

Fig 36 shows the variation of throughput when sweeping across the available SMs, allocating a fraction to Histogram and the rest to PIConGPU. A scaling of the input size is also performed starting from 12.5 million particles going up to 100 million particles. As

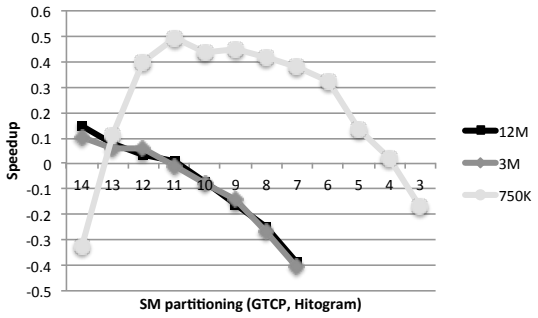

Figure 37: SM scaled throughput of GTC-P and Histogram workflow

the charge deposition calculation in PIConGPU is more compute-intensive than histogram calculation for the same problem size, the combined throughput starts to drop sharply if more than 4 out of 15 SMs in Titan's K20 GPUs are allocated to histogram. This is even more so as the problem size gets bigger (shown by the sharper slowdown for 50M and 100M). Overall, *Symphony* provides 30% better throughput than the HW scheduler for all problem sizes allocating anywhere between 2 to 4 SMs to histogram.

5.4.2.2 GTC-P and Histogram

Fig 37 shows the throughput characterization with a SM partitioning sweep for the GTC-P and Histogram workflow. The scaling is limited for GTC-P (6 million and 12 million particles) due to the memory capacity of 6 gigabytes on Titan's K20 GPUs and the problem size increasing by 4x due to quadratic dependency on radial and toroidal dimensions of the GTC-P simulation structure. The charge deposition calculation in GTC-P is nearly 2x slower (4x when normalizing to the input size) than that of PIConGPU. As a result, even with 1 SM allocated to histogram, it completes before charge deposition finishes. Thus allocating any more SMs to histogram only serves to reduce the overall throughput. In this case, *Symphony* provides 15% better throughput than the HW scheduler by allocating 1 SM to histogram.

5.4.2.3 LAMMPS and Nearest Neighbor

The scaling study varied the number of atoms per GPU from 1 million to 8 million. The two most compute-intensive phases during a time step for the above configuration of LAMMPS

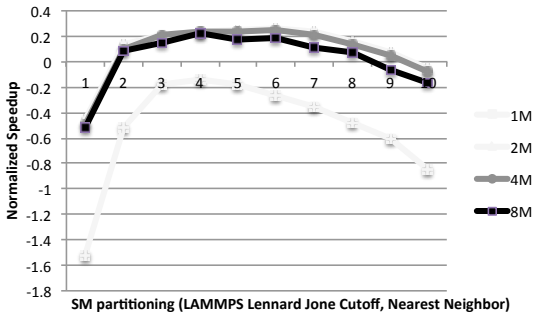

Figure 38: Scaling throughput of LAMMPS Lennard Jones with Cutoff and Nearest Neighbor workflow

is the neighbor list calculation and pair-wise force calculation, both of which run entirely on the GPU. In fact, similar to PIConGPU, this configuration of LAMMPS with the USER-CUDA package runs all computation on the GPU, including the two above steps and the time integration (NVE or micro-canonical ensemble). The pair-wise force calculation that occurs during each time step constitutes 75-80% of the running time. The neighbor list construction is 3x more compute-intensive than pair-wise force calculation but only occurs once every 20 steps in the chosen configuration. As a result, it only constitutes 10% of the overall running time. For this workflow, data consistency of atom data is automatically ensured during the force calculation phase.

Fig 38 shows the throughput characterization with SM partitioning sweep for the LAMMPS Lennard Jones with cutoff and Nearest Neighbor workflow. In this case, *Symphony* provides 20% better throughput than the HW scheduler by allocating 4 SMs to the nearest neighbor analysis.

5.4.3 Overhead

In the second set of experiments, we study the cost we pay for adding fine-grained time-slicing and SM-level space sharing capabilities to simulation kernels via added atomic operations. Kernels equipped with such functionality can dynamically scale up or scale down their SM usage and can react to the needs for running in situ analysis on the GPU. However, it is important to understand the overhead incurred to have such functionality available.

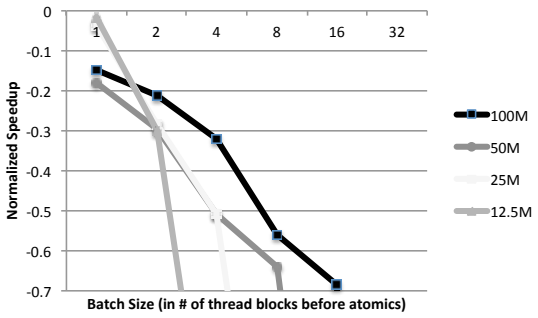

Figure 39: Software scheduling overhead of PIConGPU

5.4.3.1 PIConGPU

PIConGPU is the most well-tuned of all the scientific codes evaluated in this work. The current deposition phase in PIConGPU is similar to the other PIC code, GTC-P, in that the particle data is not modified and it constitutes 60-70% of the time spent in any time step of the simulation. However, PIConGPU performs fine-grained management of computation and data movement by launching many kernels which each handle a subset of the data. In this way, it is possible to move data around the boundary that is needed by two or more MPI ranks of the simulation running on different machines, before the last of the current deposition kernels have run. As a result, the current deposition kernel now runs over much shorter grids and load imbalance and idle cycles can be more of an issue between different supervisor kernels.

The overhead without batching for the smaller problem sizes (12.5 and 25 million particles) are low (1.5 to 3.7%) for small problem sizes but are higher (14.8 to 17.8%) for larger problem sizes (50 and 100 million particles). Serialization starts to occur at batch sizes of 2, 4, 8 and 16 for the problem sizes of 12.5, 25, 50 and 100 million particles, respectively. Even before the batch size is large enough to start causing serialization, performance starts to degrade due to the increased cost of calculating the next range of thread blocks for the 3-dimensional grid of the current deposition kernel. Overall, batching does not provide any benefits to the current deposition kernel of PIConGPU due to the small grid sizes over which it is launched and the higher cost of 3-dimensional thread block index calculation with increasing batch sizes. The overhead characteristics with different batch sizes starting

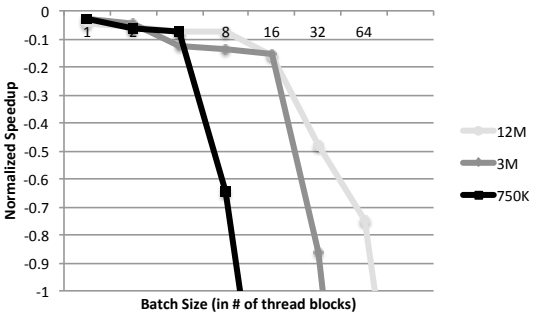

Figure 40: Software scheduling overhead of GTC-P

from no batching is shown in Fig 39.

5.4.3.2 GTC-P

Like PIConGPU, GTC-P invokes a single charge deposition kernel which constitutes 60-70% of the running time of one time step. In GTC-P the grid size of the charge deposition kernel is just over 3000 blocks since thread blocks operate on more particles. As explained earlier, small grids can quickly serialize when the batch size is increased. However, GTC-P uses a 1-D array for particle data, which lowers the cost of calculating the next thread block index in the supervisor kernel when compared to PIConGPU.

The charge deposition kernel in GTC-P has a large register footprint (182-189 registers per thread), and increasing the batch size degrades performance slightly (6-16%) before the serialization effect starts to take over at batch sizes of 4 and 16 for problem sizes of 0.75 and 3 million particles respectively. Although, we did not expect serialization to occur before a batch size of 64 for the largest problem size of 12 million particles, the performance starts to degrade from 16 and at 64, there is a sharp decrease, as shown in Fig 40.

5.4.3.3 LAMMPS

The LAMMPS Lennard Jones kernel computation takes up about 75-80% of the duration of an average time step, and is accompanied by a compute-heavy (3x longer than the pair-wise force computation) kernel that performs the neighbor list computation for each pair of atoms after a set number of time steps (20 for the default setting for 10% of the runtime). Due to the size of the pair-wise computation's application grid (hundreds of thread blocks),

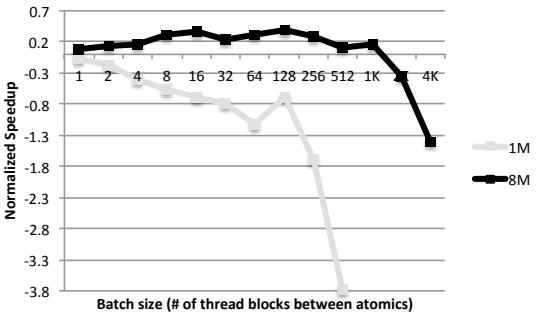

Figure 41: Software scheduling overhead of LAMMPS

this simulation is a good candidate for batching to reduce the number of *Symphony* atomics. In addition, the pair-wise kernel is 2-D. We observe that the performance for the largest problem size (8 million atoms) starting with no batching steadily improves up to a batch size of 512 thread blocks capping off at 38% (for 128 threads) as shown in Figure 41. However, when the problem size is smaller (1 million atoms), the benefit of batching gets diminished due to a smaller grid coupled with the higher cost of thread block calculation reaching serialization quickly around a batch size of 128.

5.4.3.4 Effect of batching

To understand the effects of batching on overhead, we measured three CUPTI metrics and events for GTC-P and LAMMPS. We looked at three key metrics, *gld_requested_throughput*, *gst_requested_throughput* and *atomic_transactions*. GTC-P does not see any positive effects from batching while LAMMPS shows some benefit over a range of batch sizes until serialization takes over. In both cases, the global load (and store) throughput follows the overall performance of each application indicating the overhead is due to memory bottleneck. The store throughput pattern is very similar to load throughput, so we have only shown load throughput in Figures 42 and 43.

In addition to being limited by memory bottlenecks, batching does not dramatically reduce the number of atomic transactions when compared to the existing number of atomics in kernels like charge deposition for GTC-P. By measuring *atomic_throughput*, we see that it closely follows *gld_requested_throughput* (not shown in Figure 42 due to scale). As batch

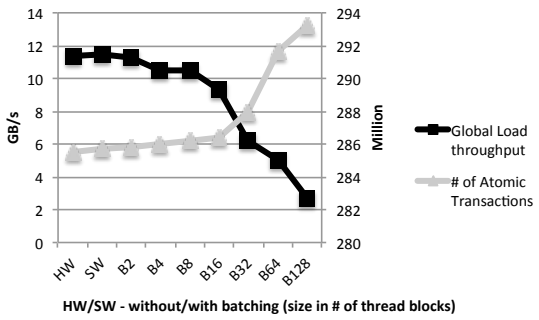

Figure 42: Effect of batching on atomics and total overhead in GTC-P

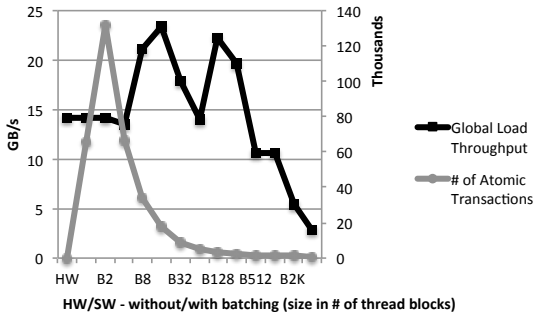

Figure 43: Effect of batching on atomics and total overhead in LAMMPS

size increases, more thread blocks generate more atomics, which eventually causes them to be serialized and increase overhead.

In contrast to GTC-P, LAMMPS does not use any atomic transactions. As a result, scheduling it with *Symphony* creates a sharp increase in *atomic_transactions* that steadily goes down as the batch size is increased. This is shown in Figure 43. It can be inferred from the measurement of *gld_requested_throughput* that the most of the overhead for larger batches is caused by a memory bottleneck and not by the added atomics for *Symphony*.

5.5 Chapter Summary

In this work, we present *Symphony*, a software scheduler for running in situ analytics jointly with scientific applications on many-core SIMD architectures, like GPUs. The key insight provided in this work is that even well-tuned GPU kernels of mission-critical scientific applications do not scale linearly to use all available cores, which allows for a valuable opportunity to co-locate analytics on a few reserved SMs.

The benefits of this approach are two-fold. On one hand, the total time to answer, which involves both running the scientific simulation and then analyzing the output data, can be significantly improved. Our approach accomplishes this by avoiding stalls in the progress of simulation due to co-scheduled analytics. Secondly, valuable cycles for transferring data back and forth between the system and GPU memory can be avoided by using the characteristics of the workflow to ensure data consistency and still allow for read-only analytics.

We also looked at mechanisms such as batching to reduce overhead of software scheduling when grid sizes are very large. In fact, unless the grid size is on the order of tens of thousands of thread blocks, batching is not necessary. This result shows that the overhead of thread block index calculation in *Symphony* is comparable to that of the hardware scheduler, which incurs the overhead of creating and destroying thread blocks several times more than *Symphony*. In summary, when the grid size is large, batching is shown to be beneficial. Future work with *Symphony* will work to create a combined superkernel for co-scheduling, and further research will be done to enable and characterize the use of more varied analytics routines.

CHAPTER VI

GPUCOFLOW : SIMULATING THE DATA TRANSFER CHALLENGES OF NEXT-GENERATION GPU CLUSTERS

With the evolution of GPUs into more generic processing engines that can handle a wide variety of data-parallel tasks, a variety of new uses and applications have arisen. Distributed GPU computing is now seeing the advent of many new machine learning applications with a push from large enterprises such as TensorFlow[1] (from Google) and CNTK[117] (from Microsoft) that are written ground-up to run on the GPU. One of the most compute-intensive tasks in machine learning is to train an artificial neural network (ANN)[20] and that is where the GPU comes into play. By running across multiple GPUs, the time to train a ANN can be reduced substantially. ANN training follows a pattern of computation and communication similar to other bulk synchronous parallel (BSP)[135] models like Google Pregel[76] and Apache Hama[119]. Distributed ANN training is characterized by periodic aggregate and broadcast operations among all the GPUs. The communication episodes are spaced out over relatively coarse-grained time intervals in the order of hundreds of milliseconds involving tens to hundreds of megabytes of data to be transferred.

Typically, intra-node data transfer bandwidth between GPUs are faster than inter-node bandwidth even with specialized interconnect such as those used in supercomputers. Moreover, new hardware interconnect technologies like NVLink[95] is likely to increase this bandwidth gap. We also observe a related trend where GPU computing clusters are supporting "dense" configurations, where individual nodes are provided with more than one GPUs. The major GPU cloud computing offerings today (Amazon[83], Microsoft[84] and IBM[53]) as well as the upcoming ones (Google[7]) have upgraded their infrastructures to support dense GPU boxes with up to eight GPUs. The latest DGX Saturn V supercomputer[103] exemplifies the trend. Summit[106], the next GPU supercomputer at Oak Ridge National Lab will also feature dense compute nodes.

Thus, the next challenge is to efficiently orchestrate both bandwidth and computation when jobs have to scale to more GPUs than there are on a single instance of these dense boxes. In the datacenter environment, the operators rely on TCP to enforce per-flow fairness. In recent times, software-defined networking (SDN) has been increasingly used ([6], [44]) to deal with congestion caused by long TCP flows conflicting on one or more links. The situation is less standardized in the supercomputing environment with vendor-specific network hardware and low-level communication libraries [8] that try to eliminate some of the overhead in going through the operating system and TCP stack. However, the goal is similar with each vendor trying to maximize the overall bisection bandwidth of the network.

Recent work has shown that "coflow"([25], [26], [112], [142]) or application-aware sharing of network resources in computing clusters is much more effective than simpler flow-level management. It can improve both the throughput of completed jobs as well as ensure that a higher fraction of jobs complete inside their deadlines. As an example, distributed ANN training can run for several hours. There is significant benefits to be had for users and providers alike by coflow scheduling if it can speedup job completion times and/or reduce fraction of jobs with missed deadlines, as the hourly rate for using these dense GPU instances are quiet high.

We believe that experimental research on coflow scheduling of distributed GPU applications can benefit if software simulation models of dense GPU clusters are available that model the network at a higher granularity than packet-level simulators such as ns-3[138]. The repetitive nature of the execution pattern of these workloads suggest that relatively coarse-grained simulation models can produce quite accurate estimates of their actual execution characteristics, which can then be used to evaluate different scheduling algorithms for various performance objectives. This work tries to address the above need and its specific contributions are as follows -

- A simulator, *gpucoflowsim*, which can be used to model next-generation "dense" GPU clusters and simulate the execution of bandwidth-sensitive jobs such as distributed ANN training. In addition, multiple coflow scheduling policies to share network bandwidth across different mixes of distributed ANN training jobs are presented. By

implementing these policies inside gpucoflowsim and comparing them with fair-share bandwidth allocation, we show that coflow scheduling can significantly improve cluster throughput by at least 38% for mix of jobs with low and high bandwidth requirements and by at least 13% even when all jobs have high bandwidth requirements.

- A set of design considerations for implementing a GPU coflow frameowrk that can act as a bridge between GPU cluster computing frameworks (like Tensorflow) and cluster resource managers (like Mesos[49]) to enable coflow scheduling.

6.1 Background

The use of ANN for machine learning tasks such as image classification[58] and speech recognition[12] have seen rapid growth in the last few years. Most of this has been fuelled by the use of GPUs that provide much higher processing power (or FLOPs) per dollar than CPUs today. Imagenet[33], considered to be the foremost visual recognition challenge in the world, started in 2010 and have seen ANN entries winning the challenge every year since 2012. The 2012 winning entry called Alexnet[67] was the first model to demonstrate the efficiency of GPUs in training ANN. Since then, models have evolved from training on single GPUs to training on multiple GPUs on the same machine and eventually, to multi-GPU, multi-machine, fully distributed GPU training (Microsoft's winning entry from 2015[48]).

6.1.0.5 Parallelization Approaches

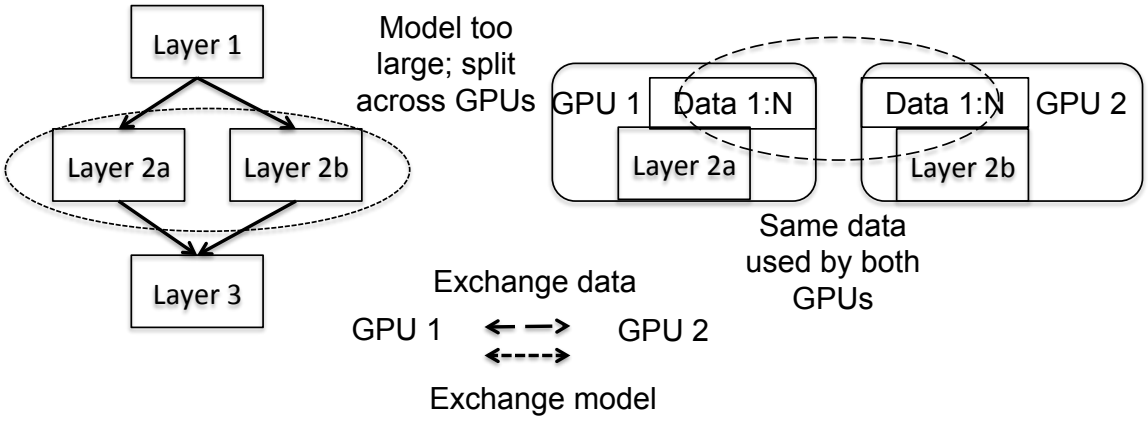

Figure 44: Distributed ANN training : Model Parallelism

There are multiple ways to parallelize ANN training. Figure 46 shows the nomenclature

for the different approaches used for distributed ANN training. *Model parallelism*[35] is useful when the number of parameters in the model is very large and it cannot be fit into the memory of a single GPU. As shown in Figure 44, a layer with many parameters, such as Layer 2, can be split into multiple layers, Layers 2a and b, placing them on two different GPUs. The same data samples coming out of the previous layer, Layer 1, are fed to both the GPUs. Once the data samples have been processed by 2a and 2b, the output from both GPUs have to be passed to the next layer, Layer 3. By updating only half the number of parameters (for layer 2) on each GPU, the data transfer necessary to synchronize the model parameters after processing a batch of data samples is also halved. Most of the ANN models today are not large enough to fall into this category. However, it may become more relevant in future for very large unsupervized learning task that involve many fine-grained features.

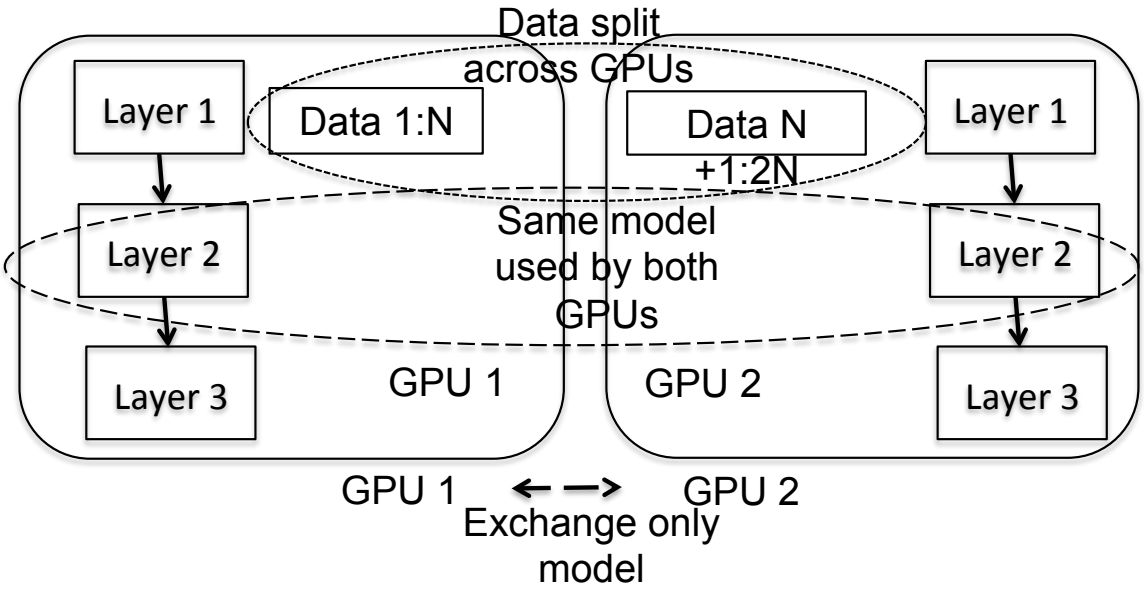

Figure 45: Distributed ANN training : Data Parallelism

Data parallelism[34], shown in Figure 45, is the more popular approach for doing distributed ANN training today. The complete training dataset is distributed uniformly across all the GPUs participating in the training. The amount of memory on a single GPU determines how much of the input data can be trained on it before new training data is fetched either from the system memory or from storage (which will cause a stall of a few seconds).

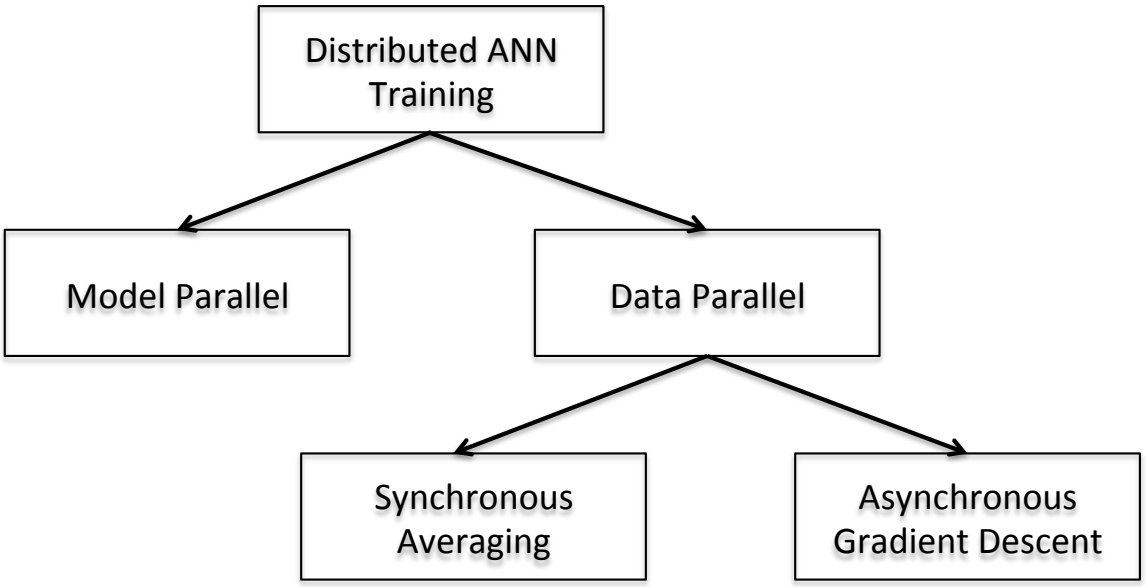

Figure 46: Distributed ANN Nomenclature

So, larger the memory, lesser the effect of stall due to data loading/unloading. Data parallel training implicitly assumes that all the parameters to describe the ANN can completely fit into the GPU memory. This is true for most of the models in use today with the largest models requiring less than a gigabyte of memory[122]. Each GPU starts off by reading the input data on its local memory in a fixed granularity of number of samples, called "minibatch"[72]. This data is used to update the parameters in the local copy of the NN model residing on each GPU. At this point, the local copies go out of synch. The models can be brought back in synch using one of two approaches, discussed next.

6.1.0.6 Synchronization Approaches

In the *parameter averaging* approach, the updated parameters from each GPU have to be first averaged and then the averaged copy made available to all the GPUs. In other words, *parameter averaging* approach is *synchronous*. After processing every N minibatches, the local copies of the model on each GPU have to be synchronized. There is some "staleness" when N is greater than 1 but there is a fixed upper bound on the "staleness". The obvious disadvantage is the reduction in throughput or number of input samples processed per second due to potential stalling of useful computation for completing the synchronization step.

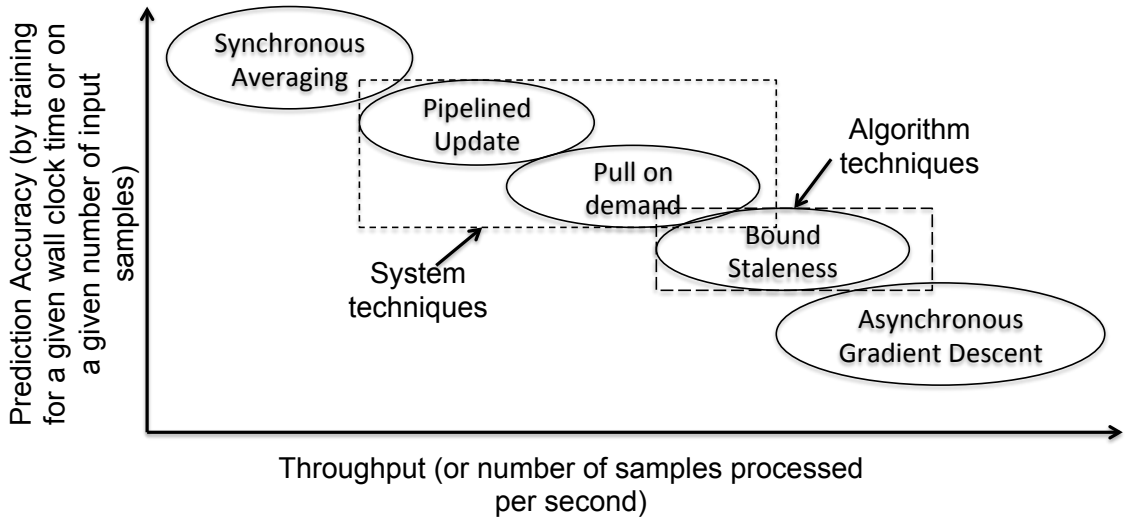

Figure 47: Distributed ANN : Throughput v/s Accuracy

In the *stochastic gradient descent*[146] or the "update based" approach, the updates in the parameters of the model in each copy is sent instead of the actual paramaters. This makes it possible to eliminate transfer of parameters that have not changed. Moreover, the updates can be send in an *asynchronous* manner without the need to stop computation on all GPUs. As a result, the "asynchronous gradient" method can improve the throughput significantly by temporarily allowing the local copies of the model on the GPUs to diverge. Unlike the "synchronous averaging" method, there is no fixed upper bound on the staleness of the gradients received by a given GPU from others. As a result, the running time to reach a specified level of accuracy of prediction using the trained model, may increase. In some cases, the desired accuracy may not be reached at all. Figure 47[123] shows the tradeoff between accuracy and throughput going from the synchronous averaging approach to the asynchronous gradient approach. There is active research[52] to find algorithms that can tradeoff the two. Gradient descent is the approach that is more widely in use today combined with optimizations to reduce the overhead of a completely synchronized update.

6.1.0.7 Bandwidth Slack

The different ANNs in use today can be broadly classified into two categories based on their topology. Feed-forward[139] is the most commonly used ANN topology where signals flow only in one direction. In feed-forward ANNs, the output of any layer of neurons do

not affect that layer. Convolutional Neural Networks (or CNNs) belong to this category. CNNs are very well suited for image classification tasks. There are some popular CNNs that are in wide use today like Alexnet[67], Googlenet[128], VGGnet[122] etc. All of them have been extensively optimized to extract the best performance on the latest GPU hardware. Residual ANNs[48] are another type that also belong to the feedforward category. Residual ANNs typically have much larger FLOPs to byte ratio than convolutional ANNs and as a result, they offer greater network bandwidth slack for distributed training.

On the other hand, with feedback ANNs, signals can travel in both directions due to loops. As earlier signals computed by a given neuron layer can be fed back into it, this category of ANNs have memory which make them very useful for time series data. Recurrent Neural Networks (or RNNs)[39] are an example of feedback based ANNs used for natural language processing tasks like speech transmission. There is a lot of ongoing work to use GPUs for training RNNs (for example, DeepSpeech from Baidu[12]). RNNs can potentially offer even more FLOPs to byte (due to cyclic execution) than CNNs and Residual NNs making them suitable candidates for distributed training. However, standard datasets and model specifications for RNNs are not yet as widely available to the research community as they are for CNNs and Residual NNs. They represent an interesting point in the design space where the network bandwidth can be oversubscribed more than they are for other ANNs and/or slower networking infrastructure could be considered.

6.1.1 Motivation

Distributed ANN training is likely to be the primary consumer of GPU compute cluster resources over the next few years. We think that their execution, communication and scaling characteristics make them a compelling case to drive research and development of application-aware network bandwidth management techniques that cater to GPU compute clusters. Next, we analyze the characteristics of these workloads and argue how application awareness in the network control plane can be used to exploit such properties.

6.1.1.1 Limited Scaling

For distributed ANN training using gradient descent, there are tradeoffs involved in choosing the appropriate minibatch size. If the minibatch size is small, the gradient descent calculated after each step is more precise. On the other hand, when the minbatch size is large, the variations seen between two successive steps would be smaller. Finally, in terms of using the FLOPs efficiency on GPUs, it is better to have larger minibatches upto the maximum available FLOPs on the GPU. For modern GPUs that can support 10 Teraflops or more, this gives quiet a large headroom for choosing a large enough minibatch. Due to the effect of multiplying the minibatch size by the scaling factor (or the number of GPUs used), data parallel distributed training ends up generating a much larger effective batch size. Due to the tradeoffs associated with choosing a batch size, it is difficult to achieve weak scaling beyond some point for these workloads. As supported by other studies in literature[27], we would observe performance characteristics of throughput and total completion time similar to the ones observed in a recent study on mini batch based ANN training shown in Figure48.

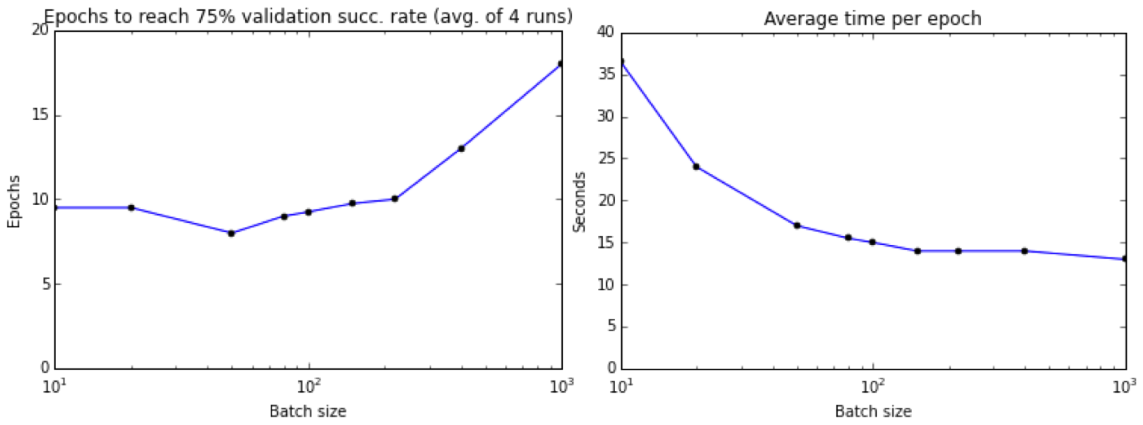

Figure 48: Distributed ANN : Limits on Weak Scaling

An *epoch* in ANN training is said to be completed when all the samples present in the input dataset has been used once. Due to the use of an iterative optimization method like stochatic gradient descent, usually it takes multiple epochs or multiple passes over the complete input dataset to achieve a prediction accuracy that is useful. Increasing the batch size is only effective upto the point when the number of epochs needed to meet the threshold does not increase. This occurs around a batch size of 100 in the above study (see chart

on the left in Figure 48). The benefits of distributed ANN training in the data parallel approach is derived from weak scaling. Distributing the samples from the input dataset across multiple GPUs makes it possible to run through an *epoch* in less time. However, the increased throughput is only going to translate into overall speedup if the number of epochs to run remain constant. This does not hold when the number of GPUs are increased beyond a certain number of GPUs and the effective batch size increases. Consequently, we observe (see chart on the right in Figure 48) that the total running time decreases up to a point and then starts to increase again. It does not make sense to scale a distributed ANN training job beyond some number of GPUs. To the best of our knowledge, we are not aware of distributed ANN training that have scaled to more than 128 GPUs[36]. In practice, the scaling efficiency diminishes rapidly going beyond 32 GPUs[52]. It would be reasonable to assume that a lesser number of "dense" (many GPU) nodes in a cluster is more suited for than a large number of "light" (single GPU) nodes would be better suited for running this type of workloads.

From a job placement consideration, this suggests that the bandwidth sharing effects of these workloads can be limited to a few racks. Distributed dataflow workloads in the CPU world, such as MapReduce[32] or Spark[141], can scale to hundreds of machines and potentially affect the entire cluster. In contrast, a distributed ANN training job that would run on a much fewer number of machines will interact with a relatively small set of other jobs. Due to this, not only scheduling but placement too can play a crucial role in driving overall performance.

6.1.1.2 GPU Utilization and Network Activity

For data-parallel, distributed ANN training, there are two key parameters that determine how utilization of GPUs are going to be affected. They are (1) the number of parameters or weights to express the ANN, and (2) the number of FLOPs to process an input sample using the ANN (in practice, samples are processed in mini batch of size greater than one).

GPUs are unlikley to be shared among multiple training jobs due to many reasons (for example, security). So, the time to train a batch only depends on (2) and is not influenced

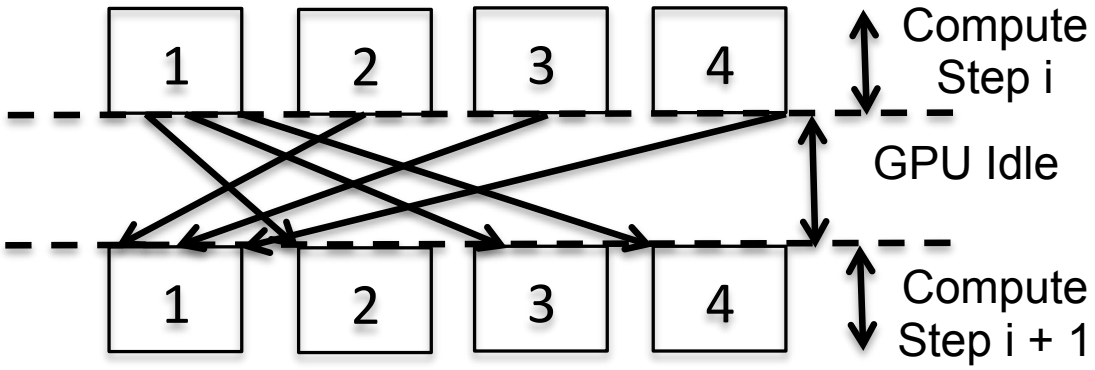

Figure 49: GPU utilization : Without software pipelining

by other co-running jobs. The training can proceed in a completely synchronized approach where a GPU does not start processing the next minibatch until the updated parameters from processing the previous minibatch have been synchronized across all the GPUs. This is shown in Figure 49 where a training job runs on four machines with some number of GPUs on each. After the ith compute step, the updated local copy of the parameters from each machine is sent to all other machines before the (i+1)th compute step can begin. In this approach, the staleness from the gradient descent is minimum. But, depending on (1) and the available bandwidth, there will always be idle cycles on the GPU during each "weights synchronization" phase. Also, any bandwidth allocation to the job, will go unused during the "weights update" phase. Bandwidth reallocation incurs some control overhead which will increase in this case.

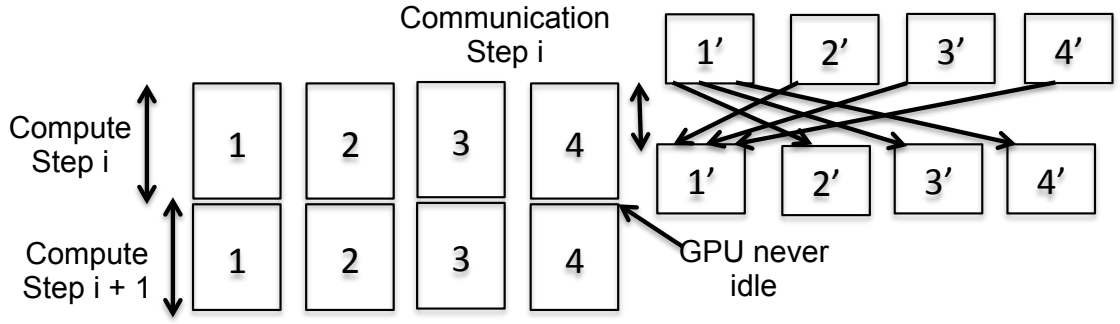

Figure 50: GPU utilization : With software pipelining - GPUs never idle

Software pipelining can be used to overlap the "weights update" and the "weights synchronization" phases of a training job as shown in Figure 50 and 51. Each GPU uses two

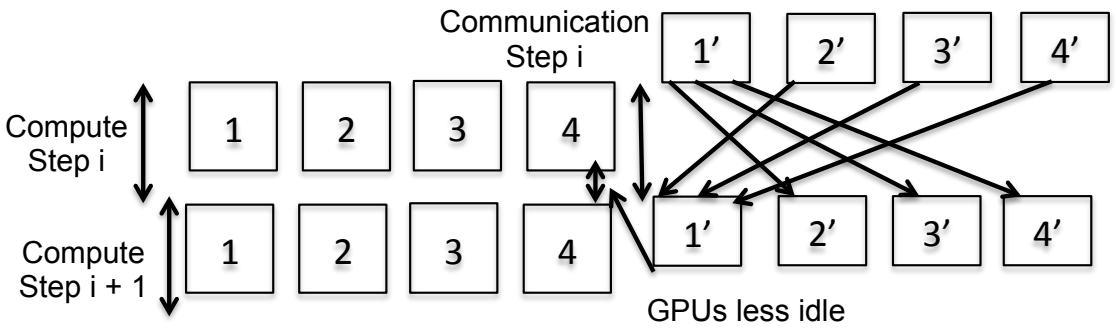

Figure 51: GPU utilization : With software pipelining - GPUs idle for less time

buffers to store the weights of the ANN. The copy in one buffer is updated when processing the current batch of input samples. In parallel, the other copy is sent out to the other GPUs. Once both of these operations complete, the buffers are flipped. This increases the staleness of the gradient descent as GPUs go out of synch by two batches instead of one. But, when (2) is high and (1) is low, it may be possible to keep the GPUs busy at all times by completely overlapping the "weights synchronization" and the "weights update" phases. This is shown in Figure 50. For example, GoogleNet[128] requires 1.5 billion FLOPs to run but has only 6.9 million parameters to transfer. Running on the latest GPU hardware, the fastest GoogleNet implementation out there today[86] takes nearly 300 ms to run the 1.5 billion FLOPs on a batch of 128 images from the Imagenet database (each image is of size 224x224). That is around ten times longer than the 30 ms it takes to transfer the 6.9 million parameters over a 10 gigabit ethernet link. In such cases, there would be intermittent periods of network inactivity from a job. On the other hand, when (2) is not high enough relative to (1), the "weights synchronization" phase cannot be completely overlapped. This is shown in 51. But even then, the time for which GPUs are idle gets reduced.

So, GPU utilization and network activity could provide a good indication of whether a job is getting its requested bandwidth, and such information can be used by the network bandwidth scheduler to enfore better sharing.

6.2 Simulator Design

In this section, we will describe the detailed design of our Gpu Coflow simulator. It consists of four primary functions which are -

1. Workload generation
2. Assignment of jobs to machines
3. Aggregation and broadcast scheme
4. State machine to track job progress
5. Scheduling policies for bandwidth sharing

6.2.1 Workload generation

This area of distributed ANN training is still in its infancy. Therefore, production traces are hard to find unlike other distributed infrastructures such as Hadoop, Spark etc. But with the rapid growth in the user base of deep learning frameworks like Tensorflow[1] and their increased footprint on cloud computing resources, it is only a matter of time before researchers would gain access to such production traces. Until then, statistically generated traces have to be used. These jobs typically run for hours which gives some margin for approximation as to when a particular job arrives. If new jobs continue to arrive and the cluster size is fixed, there has to be admission control soon after the cluster is oversubscribed. Due to their long running nature, it is not possible to meet any reasonable service guarantees for individual jobs by oversubscribing the cluster too much. When the number of jobs are low, it should be possible to consolidate resources in a way such that the operating state is close to fully subscribed.

Due to the above characteristics, we have made the following two assumptions in our simulation model. First, all the jobs have already arrived before the simulation starts. Second, there are just enough jobs available to fully subscribe the available resources. The situation that occurs when there are more jobs than available resources is interesting because the jobs have to be cycled through alternating execution and waiting phases. There exists a resource allocation problem in such a scenario where the choice of jobs to run together could be critical to overall performance. This is a problem orthogonal to the scheduling problem for efficient bandwidth sharing that we address in this work and so is out of scope.

For describing a workload in the simulator, we use an entity called *job*. A job is characterized by three parameters, namely, (1) the number of FLOPs to process a batch of training data, (2) the number of bytes to transfer for each batch processed, and (3) the number of machines over which the training data is distributed. (1) depends on the GPU hardware and the implementation of the ANN. Most (if not all) frameworks call into the GPU vendor's implementation of the numerical libraries (for example, cuDNN[100]) implementing the underlying mathematical operations (namely, BLAS[98] and FFT[101]) tailored to the requirements of standard deep learning algortihms. So the computation time to process a batch, that is, the "weights update" phase with a given ANN primarily depends on the GPU hardware. For a given cluster, the GPU hardware is fixed. (2) depends on the number of weights to represent the ANN. As the training process consists of processing several batches of the same size, both (1) and (2) can be inferred from run-time profiling. Setting up a distributed ANN training job requires application-dependent placement of input data on the participating machines. There is ongoing effort[29] to integrate deep learning frameworks with the Hadoop ecosystem which will enable system-level management of training data using HDFS. In future, it may be possible that for a given job, the system chooses the number of machines to run the distributed training on (that is, (3)). This gives the additional flexibility to dynamically scale tasks based on the demand for network bandwidth. However, until such mechanisms become available, we assume that (3) would be specified by the user. Due to the scaling limits of distributed ANN training jobs as discussed in 6.1.1.1, we assume that specified value of (3) is the upper limit.

Next we describe how the specification of the cluster is provided to the simulator. There are four parameters - (a) the number of machines on a rack, (b) the intra-rack bandwidth, (c) the number of racks, and (d) the inter-rack bandwidth. The intra-rack and inter-rack networks are abstracted as fully-interconnected non-blocking switches. This simplifies the implementation a great deal yet represents a reasonable approximation because we have seen evidence that datacenter networks are evolving towards software-controlled topologies like VL2[44] that can provide full bisection bandwidth between any two endpoints on the network.

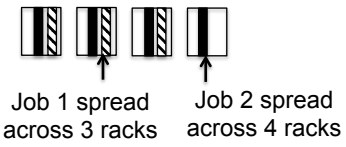

Figure 52: Always spread across racks

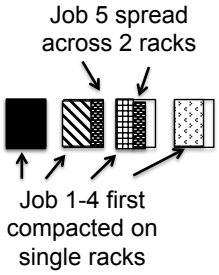

Figure 53: First compacted then spread

6.2.2 Assignment of jobs to machines

There are two aspects to consider for placing jobs. First, how is a single job allocated to machines and racks? Second, how are neighbors of a job chosen? Optimal placement algorithms that give the best performance (for example, throughput) are unlikely to be of polynomial complexity (rigorous proof is outside the scope of this paper). We provide a qualitative discussion of the flavor of the different placement algorithms and their effect on network bandwidth sharing. When a single job is allocated to machines and racks, the *always spread*(Figure 6.2.2) and the *first compact then spread*(Figure 6.2.2) placement policies represent two ends of the spectrum. In the *always spread* approach, each job is spread over as many racks as possible. By doing this, intra-rack traffic due to this job is minimum (or zero) while the inter-rack traffic is maximum. The downside is that any effective scheduling algorithm requires a lot of global information.

On the other hand, in the *first compact then spread* approach, a job is assigned a rack that has enough free machines, if possible. Otherwise, it is spread across the least number of racks. In contrast to the *always spread* approach, this approach tries to minimize inter-rack traffic. For jobs that are fully contained within a single rack, the scheduling algorithm can function effectively using entirely local information. However, how spread out the other jobs are depends on the the mix of how many machines each job needs. Again, it may involve a

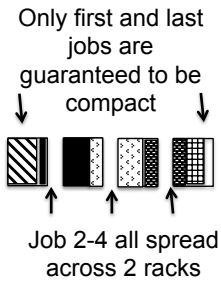

Figure 54: Compacted or bounded spread

lot of global information exchange to effectively schedule some of the jobs.

As we have discussed earlier, distributed ANN training jobs are likely to scale out to tens of GPUs but not hundreds (64 to 128 GPUs, at most). For dense nodes with 4 to 8 GPUs, that ranges between 8 and 32 machines. With the cooling capacity likely to be the most critical factor in determining rack sizes (30 KW per rack), we anticipate that between 10 to 20 machines would be the limit a rack can accomodate[87]. Therefore, most jobs will fit inside one rack with only a handful of jobs spanning to more than one rack. In view of the above observations, we posit the *compact or bounded spread*(Figure 6.2.2) approach to be a better fit where a rack is completely filled before another rack is chosen. By doing this, inter-rack traffic due to any job spanning more than one rack, is intuitively lesser than the *first compact then spread* approach. Effective scheduling algorithms for such a placement strategy will require a constant amount of global communication between racks. A rigorous proof is outside the scope of this work.

The order in which jobs are assigned to machines decides which jobs contend for the bandwidth inside a rack and between racks. An optimal assignment is again a task that is intuitively not of polynomial complexity. Again, we consider some of the policies and how they would affect performance. If all jobs with the same type of ANN are ordered together, all the jobs on a rack would have the same bandwidth requirement which renders a scheduler pointless. To ensure the maximum diversity of traffic on each rack, we chose to generate jobs by cycling in a round robin basis through all possible types of ANNs specified as input to the simulator. A tradeoff between the two might follow a uniformly random distribution for choosing the next job.

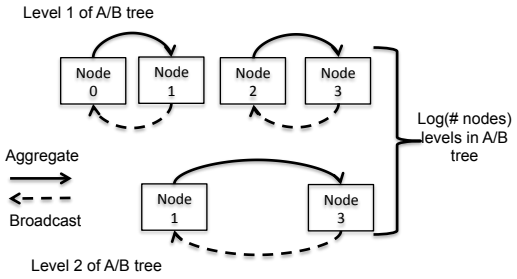

Figure 55: Aggregation and broadcast using a tree

6.2.3 Aggregation and broadcast scheme

Based on the number of machines used for distributed training as well as number of weights to represent the ANN, there exists multiple schemes to orchestrate the synchronization across all the machines of a job. If a tree-like hierarchical synchronization is performed across N machines, there would be logN levels of the tree. If the training is distributed to a large number of machines and the number of weights of the ANN being trained is low, the tree-based synchronization is very effective. To synchronize D bytes of weight across N machines given a bandwidth of B, it requires logNx(D/B) units of time. This is shown in Figure 6.2.3.

On the other hand, the value of N is likely to be low due to limited weak scaling property of distributed ANN training. Also, the minimum number of weights in many of the standard ANNs are hundreds of megabytes which makes the overhead of filling the pipeline a small fraction of the total transfer time. To syncrhonize the same D bytes of weights among N machines, transferring in units of d bytes each, it requires 2xNx(d/B) units of time to fill up the aggregation pipeline at the start and empty the broadcast pipeline at the end of the synchronization. Asssuming D is much larger than d, the (D-2d)/B units of time required for the rest of the transfer dominates. Therefore, a pipelined synchronization is more effective with not too many stages in the pipe to fill. This is shown in Figure 6.2.3.

The simulator models a pipelined data exchange. For all the machines of a job, the weights are received from another machine in the same job that has the index one less than the current machine's index (except the machine with the least index). The received weights are aggregated with the local weights on the machine. Then, the aggregated weights are

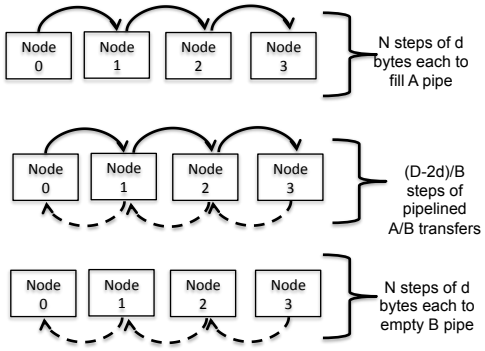

Figure 56: Aggregation and broadcast using a pipeline

send to another machine which has the index one more than the current machine's index (except the machine with the highest index). In total, there are 2N - 2 aggregation traffic flows for a job running on N machines. Once the weights have been aggregated by the last machine, they need to be braodcasted back to every other machine. The broadcast phase is carried out similar to the aggregation phase. Each machine (excluding the last one) receives a broadcast from another machine with an index that is one higher than that machine. At the same time, every machine (excluding the first one) sends a broadcast to another machine with an index that is one lower than that machine. As a result, there are also 2N - 2 broadcast flows. Furthermore, the aggregate and the broadcast phases can be overlapped to exploit the full-duplex bandwidth of the ethernet switches (or infiniband switches). For dense nodes with multiple GPUs, aggregate and broadcast phases are needed to synchronize the weights between the GPUs on a single machine. By overlapping the aggregate and broadcast phases using software pipelining, the intra-node synchronization can efficiently utilize the full-duplex data transfer capabilities of PCI-express (or NVLink) that interconnects GPUs on a single machine.

6.2.4 State machine to track job progress

After a pool of jobs have been created and each job has been assigned machines, the next question is to simulate the distributed execution of the job using the assigned machines. For this, we introduce the next level abstraction called *machine manager* which is responsible for all activites on a machine and for synchronizing with other machine managers after execution

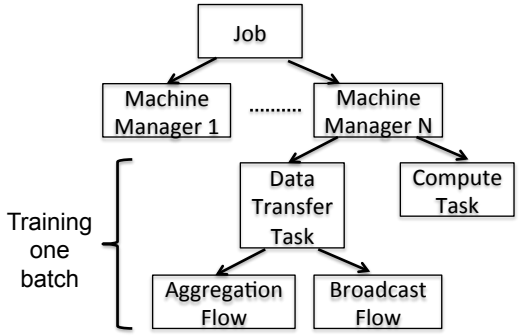

Figure 57: Components of a job to process a single batch of training data

of each minibatch. Next comes a *task*. Each machine manager consists of multiple tasks. In an actual implementation (for example, Tensorflow), a task might be a thread responsible for managing a particular GPU or sending/receiving the ANN weights to/from other machines. In our simulator, there are only two types of tasks - a *computation task* and a *data transfer task*. A computation task is modeled simply by advancing time according to the job's ANN. The data transfer tasks are more intersting being responsible for synchronizing the weights with data transfer tasks on other machines, either on the same rack or between two racks. A task consists of one or more *flow*s. A flow, in our simulator, is the actual consumer of available bandwidth inside a rack and between racks. Flows can be used either as an *aggregation flow* or as a *broadcast flow*, in both tree-based and pipelined synchronization mechanisms. This is shown in Figure 6.2.4.

The simulator proceeds in discrete time ticks during which the flows are updated. The remaining bytes to be transferred on a flow is decreased by the actual bytes transferred since the last tick which in turn depends on the bandwidth allocated to this flow. When the remaining bytes on a flow reaches zero, the flow is considered complete. A task is considered complete When all flows in the task have been closed. New tasks can be continuously started for a job as long as the simulation is running. In practice, at some point, a job will be considered complete when the training accuracy reaches some desired threshold. But a job typically runs for several hours before that happens. Our goal is to simulate up to a point where the throughput of the cluster has reached a quiescent state. Usually, this happens within a few minutes. So, it is a reasonable assumption that all jobs would have

an unlimited supply of new tasks over the length of the simulation.

Two types of events can cause the scheduler to be invoked - (1) when tasks are created, and, (2) when tasks are completed. Due to the bulk-synchronous nature of distributed ANN training, new tasks can only be created when all previous tasks belonging to the same job have completed including the compute tasks. When such an event occurs, the number of flows contending for the intra-rack and the inter-rack bandwidth increases. This calls for a redistribution of bandwidth whereby existing flows need to give up a part of their allocated bandwidth to the flows that were just created. On the other hand, when tasks complete, their allocated bandwidth may become available for use by flows of other tasks. This occurs when other data transfer tasks for the same job have not closed possibly due to residing on other racks or the compute tasks have not completed. In such a case, extra bandwidth becomes available which can be distributed among the existing flows.

6.2.5 Scheduling Policies for bandwidth sharing

Once the state machine outputs a task creation or completion event, the scheduler in the simulator gets triggered. It takes as input 1) the continuing flows, 2) the created flows, and 3) the completed flows, for each rack as well as between racks. When new flows are created, the scheduler needs to assign them bandwidth by reducing the bandwidth allocated to the continuing flows. Likewise, when flows complete, the scheduler needs to assign the freed up bandwidth to continuing flows or newly created flows, if any. Computing an optimal allocation for any performance objective (for example, maximize total progress of all the jobs) is intuitively hard (not of polynomial complexity). Moreover, it would require exchange of a considerable amount of control information between racks. We explore two scheduling policies for effective reallocation of bandwidth based on the characteristics of the ANNs constituting the job mix.

6.2.5.1 Least requested bandwidth first (LRBF)

Jobs with ANNs that have fewer weights to transfer could process a batch even with less bandwidth. By favoring flows from such jobs, the overall throughput of the cluster can be improved albeit at the cost of reducing fair progress. Every time new flows are created, all

flows are ranked according to the number of bytes they need to transfer. New flows would need to tranfer all the weights of the corresponding job's ANN. Continuing flows would have transferred some of the weights already. For each flow, the requested bandwidth is calculated from the number of bytes left to transfer and the remaining time left to complete the compute tasks of the job this flow belongs to. Bandwidth inside each rack and the inter-rack bandwidth is allocated to flows in the increasing order of their requested bandwidth.

When most of the jobs train ANNs with hundreds of millions of weights (for example, Alexnet, VGGNet, Overfeat[120]), there are many jobs with flows that do not receive the requested bandwidth. Compared to a fairshare distribution of bandwidth based just on the number of flows, the LRBF policy can be more effective in reducing the number of jobs that do not receive their requested bandwidth. That is, it can help to decrease the margin by which a job is slower in processing a batch due to not getting its requested bandwidth.

6.2.5.2 Least available slack first (LASF)

A job's *slack* is calculated by taking the difference of the time it would take for the flows of the job (on the given rack) to send their remaining bytes at their currently allocated bandwidth and the time remaining to complete the computing tasks of the job. Whenever flows of a job that has ongoing computing tasks complete, all the continuing flows that have a negative value of slack are ranked in ascending order (most negative to least negative). A negative value of the slack indicates that those jobs have flows that are not going to complete before their computing tasks are done. If extra bandwidth is allocated to such flows, that should help. Jobs with negative slack are selected in their sorted order and they are allocated as much bandwidth as required to raise their slack to zero. This is continued until all the extra bandwidth available from the completed flows have been used. If there are no flows (of any job) with negative slack, it indicates that the extra bandwidth is not useful for improving the throughput from the current rack. In that case, it is distributed equally among other flows of jobs on the same rack that are fully contained in the rack.

In contrast to the previous scenario where most of the jobs have hundreds of millions of weights in their ANN, there are other ANNs([48], [128]) with hundreds of thousands

up to a few million weights. When the job mix is largely constituted by such jobs, most jobs will complete their data transfer tasks before their computing tasks have completed. Instead of using a fairshare policy to distribute bandwidth recovered from early completion of data transfer tasks, the LASF policy can help to make sure that only jobs that need extra bandwidth get it.

6.3 Designing GPU coflow middleware

In this section, we look into some of the design tradeoffs that have to be considered when building a middleware framework for managing GPU coflows. The different controllers to arbitrate network bandwidth are shown in Figure 6.3.1.

6.3.1 Multi tenancy vs. single job per machine

As the individual nodes are likely to be dense with up to four or eight GPUs, it is worth debating the possibility if a node could be shared by more than one job at any given time. The main advantage is the flexibility to support more user jobs at the same time even if some jobs only need a few GPUs. With the maturity of container-based isolation mechanisms[124], the security risks of a job accessing GPUs not allocated to it can be eliminated without any noticeable overhead (if the same had to be enforced through hypervisor-based isolation[14]).

For intra-node coflow management, the resource under contention is the PCIe (or NVLink) bandwidth for synchronizing between the GPUs on the node. The only platform-independent way to use the PCIe bandwidth from/to GPUs is by going through the GPU runtime (and the driver). This is challenging because the way applications use the memcpy API provided by the GPU runtime is a lot different from their use of network send/receive API. For example, application-specific batching of small transfers is quiet common. Any middleware layer on top of the GPU runtime to manage the bandwidth use through the memcpy API has to also consider other aspects of the runtime which soon gets tied up deeply with application semantics. Considering the possible complexity to build PCIe bandwidth management middleware, it seems more reasonable to design for a scenario where all GPUs on a node would be allocated to a single job.

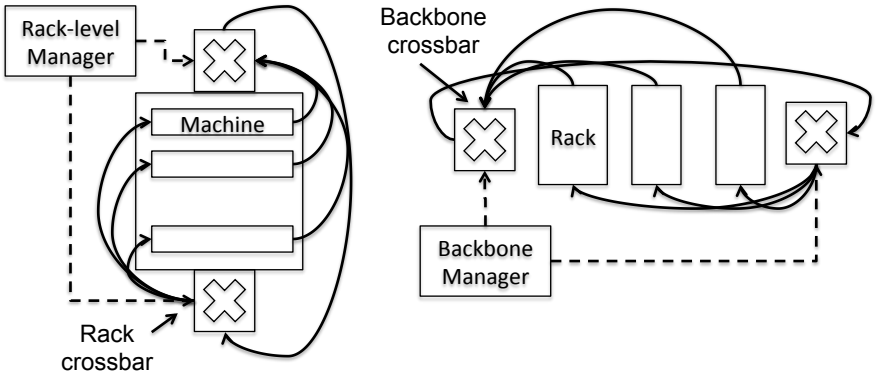

Figure 58: GPUCoflow controllers

6.3.2 Manager per job vs. manager per machine

All flows belonging to a job constitute a collective flow (or a "coflow") and treating them as a single entity for the purposes of bandwidth allocation makes sense given that they need to be synchronized after each batch. For a job that fit completely inside a single rack, the per machine managers could send their bandwidth request along with an identifier for the job to a *rack-level manager* that arbitrates the bandwidth allocation inside the rack. The complication arises due to the fact that some jobs will span across two or three racks when using the *compact or bounded spread* placement policy, and possibly more for other placement heuristics. One option is to restrict all coflow decisions within a single rack.

The downside of not considering bandwidth request from a job on another rack is that the bandwidth allocated to the job may turn out to be more than or less than what the job needs. Due to the synchronous nature of distributed ANN training jobs, a job completes processing a batch only when its slowest flows complete. Any extra bandwidth if there are slower flows or bandwidth such that the flows on the given rack are the slowest, may diminish the overall throughput. By having a manager for each machine on the rack and disregarding the spillover machines on the other racks, the complexity of the rack-level scheduler is greatly simplified. Even if more globally cognizant scheduling policies need to be enforced, the control flow between rack-level schedulers can remain decoupled from the actuation logic in a rack that enforces the scheduler's decision.

6.3.3 Overlapping vs. spread out inter-rack traffic

When jobs span across multiple racks, there would be flows between machines on different racks in addition to flows between machines on the same rack. Inter-rack flows are difficult to handle because they will affect the throughput of the racks at either ends. If the inter-rack flows for a given batch are allowed to run concurrently with the intra-rack flows, they could provide an extra layer of scheduling control on top of the rack-level schedulers to affect the overall throughput of the system. At the same time, globally optimal scheduling decisions are hard.

On the other hand, by serializing the intra-rack and the inter-rack aggregation/broadcast phases, the inter-rack scheduler can be made much simpler. In this case, the inter-rack bandwidth is used by different jobs in short bursts. At any given instant, only a few jobs use the inter-rack network and all the jobs have similar scheduling objective, that is, complete the remaining inter-rack data transfer as fast as possible. This is because the intra-rack data transfer activities of the jobs have already completed. If they were allowed to overlap, sometimes it may occur that finishing the inter-rack transfer in the shortest possible time is unnecessary because the intra-rack flows are slower. Whatever be the flow-control mechanism, all inter-rack bandwidth requests from a rack should be routed to a rack-level agent. Each such agent would then forward the requests to a *backbone manager* that arbitrates inter-rack bandwidth among the requesting jobs.

6.4 Experimental Evaluation

In the evaluation of the bandwidth scheduling policies for GPU coflows, we investigate three different aspects of their effectiveness.

- How much is the relative speedup compared to fairshare bandwidth sharing in terms of the total number of batches processed spanning across all the jobs?
- How much is the performance gain dependant on the characteristics of the ANNs constituting the job mix?

To evaluate the LRBF policy, we considered ANNs that have more than a hundred megabyte of weights. The most well-known convolutional ANNs along with their compute time for a batch of 128 Imagenet size (224x224x3) samples on a Maxwell Titan X GPU are shown in Table 3. The fastest reported 32-bit floating point implementation is used for reference. Sources are [24] and [59]. 10 Gigabit Ethernet switches are used to model the rack-level and backbone switches.

Table 2: ANNs with high number of (>50M) weights to train

Name	Number of weights (in millions)	Compute Time (in milliseconds)	Comments (fastest implementation)
Alexnet	61	80	Tensorflow
VGGNet	144	320	Neon
OverFeat	138	210	Neon

The first set of LRBF experiments uses a mix of Alexnet and VGGNet training jobs. Two configurations are evaluated, eight and sixteen GPUs respectively, that the maximum number of nodes that each job can scale up to. Higher scale up limit results in more jobs that span across racks. The increase in throughput of total number of batches processed relative to a fair share bandwidth allocation among active flows is shown in Figure 59, for increasing size of the cluster. The improvement in throughput ranges from 31-44% with the configurations that allow up to a maximum of eight nodes per job, giving better performance for all sizes of the cluster.

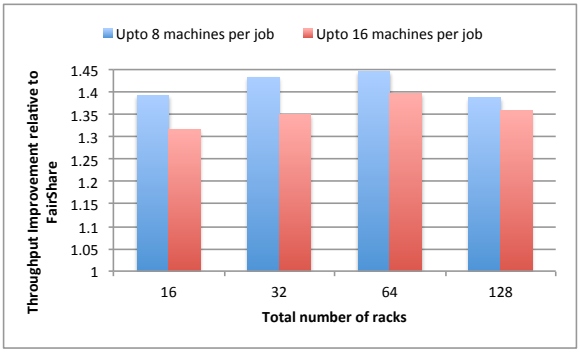

Figure 59: LRBF throughput : Mix of ANNs with low and high number of weights

In the second set of LRBF experiments, we select a mix of VGGNet and Overfeat jobs where both the ANNs have high bandwidth requirement. As expected, the improvement in throughput is lower in this case ranging between 13% and 33%. Interestingly, the configuration that allows up to sixteen machines per job, performs better for this mix. By having more jobs spread out across racks for the sixteen machine per job configuration, there are more stalls due to cross-rack synchronization which helps to free up bandwidth in a rack and let the jobs that reside on single racks make better progress.

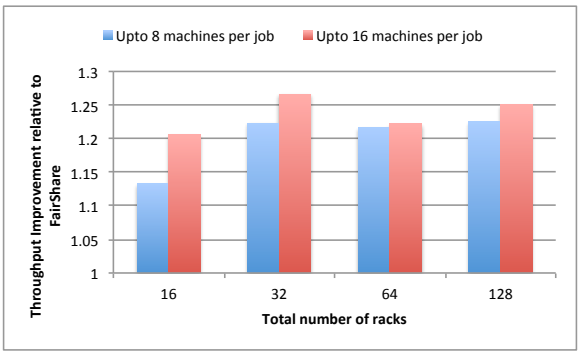

Figure 60: LASF throughput : Mix of ANNs with only high number of weights

To evaluate the LASF policy, we considered ANNs that have less than a hundred megabyte of weights. Googlenet and the Resent variants are the popular convolutional ANNs belonging to this category as shown in Table 3. Due to the significantly lower bandwidth requirements of the jobs in this case, we have used 1 Gigabit Ethernet switches to model the rack-level and backbone switches for the LASF experiments.

Table 3: ANNs with low number of (<50M) weights to train

Name	Number of weights (in millions)	Compute Time (in milliseconds)	Comments (fastest implementation)
GoogleNet	5	470	Torch
ResNet-50	30	1280	Torch
ResNet-100	40	2000	Torch

We used a mix of Googlenet and Resnet-50 jobs for our first set of experiments. Googlenet has relatively lower computation time than Resnet-50. Both have less than 50 million wieghts to synchronize. The throughput improvement ranges from 1.78x to 2.08x the

throughput of fairshare bandwidth allocation. The configuration with jobs that can scale out to a maximum of 16 machines gives slightly better throughput for all rack sizes. Due to the wider difference in bandwidth requirement of the two types of jobs and when jobs are more spread out across racks, the stalling of a few Resnet-50 flows on any rack enables to raise the throughput by allowing a greater number of Googlenet jobs to make progress.

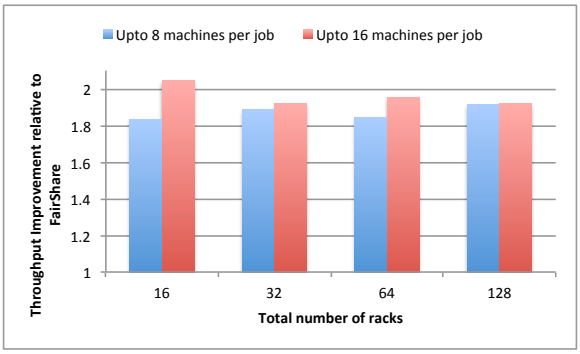

Figure 61: LASF throughput : Mix of ANNs with low and high FLOPs to compute

In the next set of experiments for LASF, we used a mix of Resnet-50 and Resent-100 jobs, both of which have very high computation time and relatively few weights to synchronize. Similar throughput improvements ranging from 1.84x to 2.05x are obtained in this case, too. However, in this case the more compact configuration of a maximum of 8 machines per job performs slightly better. The stalling in the spread out configuration cannot be offset by higher progress per rack as both jobs have similar bandwidth requirements.

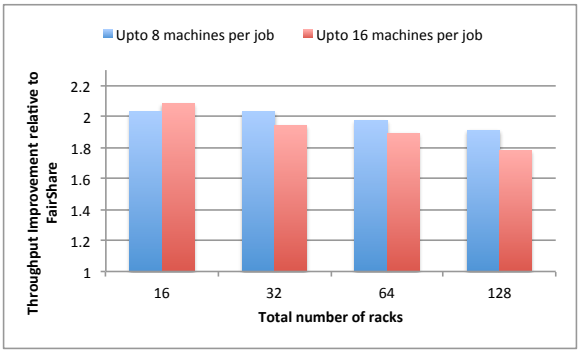

Figure 62: LASF throughput : Mix of ANNs with high and high FLOPs to compute

6.5 Chapter Summary

In this chapter, we have looked at the problem of network bandwidth sharing in dense GPU clusters that are fast becoming the infrastructure of choice for running distributed GPU

applications like training artificial neural networks that takes hours to complete even with the huge computing power of GPUs. In such a scenario, it is critical that the network is shared appropriately among the co-running jobs to sustain a high throughput at which batches of input samples are processed by all the training jobs.

We exploit the repetitive nature of these workloads to build a simulator, *GpuCoflowSim*, that captures the essence of their execution patterns without going into the excessive details of standard packet-level network simulators, allowing us to model substantially large clusters over time windows of several minutes. We have proposed coflow aware scheduling policies to manage network traffic from the different jobs and evaluated them on *GpuCoflowSim* to demonstrate their effectiveness over application-agnostic fair share bandwidth sharing policies. Finally, we present a qualitative discussion on the different design tradeoffs that are essential to consider when building a middleware framework for managing GPU coflows in a dense cluster environment.

CHAPTER VII

CONCLUSIONS, RECOMMENDATIONS AND FUTURE DIRECTIONS

GPGPUs are useful for many types of compute-intensive workloads from scientific simulations to cloud-focused applications like machine learning and graph analytics. However, unlike CPUs they do not allow for software-controlled sharing of resources. This leads to underutilization, unfair use and reduced programmability. This thesis looks at three different areas, 1) in situ analysis in scientific workflows, 2) multi tenancy in cloud computing environments, and 3) network sharing between evolving distributed GPU frameworks. The thesis presents four distinct software-scheduling based constructs to handle problems in each of these spaces.

First, the thesis looks at Landrush, an idle cycle scavenging approach for GPUs to improve time to answer in scientific workflows by running data analysis in situ with controlled interference due to co-location.

Second, the thesis presents GPUShare, which enables sharing of GPUs between long-running cloud workloads helping to reduce cost of usage by ensuring resources are fairly shared while ensuring that standalone execution remains unaffected.

Third, the thesis demonstrates Symphony, a software-supervised GPU scheduler that trades off the low overhead of hardware dispatching and the runtime responsiveness of software scheduling to improve time to answer for such scientific workflows that do not afford idle cycles.

Finally, the thesis talks about GpuCoflow, a novel approach to network sharing between evolving distributed GPU computing frameworks that considers and application's computing and data transfer characteristics to ensure increased overall throughput compared to traditional network scheduling approaches that are geared towards providing high bisection bandwidth.

In summary, this research puts forth compelling scenarios to demonstrate how GPGPU computing is managed at a system level and then presents concrete mechanisms with accompanying policies to enhance the manageability both in terms of how computation cycles on the GPU gets used as well as how data gets moved from/to the GPU with or without involving the network.

This thesis has covered several aspects of GPGPU scheduling with regards to both computation and data movement. During the course of dealing with the current GPGPU system stack (both hardware and software), we encountered multiple interaction scenarios between applications and the system stack that could be better handled if future GPU platforms are equipped with better system support. Below is a list of a few important recommendations that could enhance the system-level manageability of future GPU platforms.

- **Programmability in thread block dispatching** The current thread block dispatching mechanism in GPUs is completely hardwired. In this thesis, we have explored ways to add flexibility to how thread blocks are dispatched by diminishing the role of the hardware thread block dispatcher. This could benefit further if some level of programmability is provided to control the behavior of the thread block dispatcher that is exposed at the runtime layer above the driver. For example, the thread block dispatcher could be instructed to issue thread blocks to an SM only when a majority of thread blocks that were previously running have completed execution. This can help in a more contiguous placement of thread blocks onto SMs thereby reducing memory divergence.

- **Online tracking of variability of running times across thread blocks** Increased flexibility to dispatch thread blocks onto SMs calls for online feedback as to how good the chosen dispatching policy has been and accordingly take actions to improve the dispatch throughput. Profiling information detailing when each thread block was dispatched and which other thread blocks it was colocated with at relatively frequent intervals can be very helpful to adjust the timing and placement policies. The collection interval of such profiling data would also be critical in order to limit

overhead. Interrupt handlers inside the hardware thread block dispatcher appears to be a suitable site to install such functionality.

- **Online tracking of memory access window** This thesis has explored ways to achieve efficient multi-tenancy on GPUs. One of the key hindrance to multi-tenancy on GPUs is the limited support for memory oversubscription on GPUs. Most GPU applications perform their memory allocations up front as a result of which if the sum total of the memory footprint of two applications exceed the physical memory on the GPU, some of the allocations need to be moved to the system memory if the applications are to co-run on the same GPU. Compiler analysis can provide some insight on the range of memory accessed by each thread block but not all. Even if paging is supported in future GPUs, online profiling information about memory access ranges by thread blocks would be extremely useful. In conjunction with a programmable DMA engine that can coordinate with the thread block dispatcher at runtime, mechanisms that implement application-aware paging can do much better than application-oblivious paging that takes care of only the memory access datapath of warp-level instructions.

- **Program loading support** Loading a program onto the GPU today happens completely via the driver and fixed-function hardware. Consequently, any mechanism to enable persistent scheduler threads on the GPU has to implement tedious workarounds of stopping and restarting the scheduler threads if new programs need to run. For some usecases like the the in situ scientific analysis workflows, this is not a show stopper. However, for an environment consisting of many short-running jobs that arrive dynamically, the associated overhead can be prohibitive. If future GPUs support privilege levels, it can support higher privilege GPU kernels that can have memory execute permissions in addition to read and write permissions. Such kernels can then be used to perform loading of lower privilege application kernels and as a result, persistent scheduler threads on GPUs can work similar to the operating system's scheduler for multi-core CPUs.

- **Application-aware system control** Training deep neural networks have attracted the most attention in recent times due to the huge reduction in time required to train them using GPUs. Graph processing workloads appear to be very promising too. Iterative graph workloads such as different link analysis algorithms, e.g. PageRank, HITS, SALSA, Who-To-Follow and others look set to be the immediate beneficiaries with new dense GPU platforms offering up to 24GB of memory per GPU and up to eight GPUs interconnected at near memory speeds. It is possible to analyze a reasonable subset of a massive graph dataset like the Facebook social network (one trillion edges) without sending a single network packet. Even considering the slow PCIe bandwidth to access system memory, benchmarking studies have shown at least 2x speedup with GPUs over CPU throughput. Despite local variations due to what part of the data gets accessed, these workloads exhibit distinct longer-term repititiveness in their execution characteristics. Moreover, these workloads run long enough such that their execution characteristics may be learned and used to steer subsequent execution. Most system control mechanisms in the CPU world are less fortunate being bound by stricter limitations to cater to more generalized execution behavior. Without such stringent constraints in designing system software for GPUs, we have exploited application-awareness in all of the components in this thesis. It would be very useful if future GPUs can come up with standardized interfaces to exchange this information between the application and the system software in addition to what already exists at the hardware level, today.

If some or all of these recommended support becomes available in future platforms, it would only assist in further proliferation of GPUs in environments or application areas where their use is still limited. Some of this is envisoned in the future work, described next. At one end, future work will look at other big data frameworks to understand if the enhanced manageability principles can be used by such frameworks to reap the benefits of GPGPU computing. At the other end, it will be interesting to explore the least resistance path for cluster management frameworks to abstract the principles contributed in this thesis and export them to higher layer frameworks.

REFERENCES

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P. A., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X., "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pp. 265–283, 2016.

[2] Abbasi, H., Eisenhauer, G., Wolf, M., Schwan, K., and Klasky, S., "Just in time: adding value to the IO pipelines of high performance applications with jitstaging," in *Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing, HPDC 2011, San Jose, CA, USA, June 8-11, 2011*, pp. 27–36, 2011.

[3] Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K., and Zheng, F., "Datastager: scalable data staging services for petascale applications," in *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, HPDC 2009, Garching, Germany, June 11-13, 2009*, pp. 39–48, 2009.

[4] Adriaens, J., Compton, K., Kim, N. S., and Schulte, M. J., "The case for GPGPU spatial multitasking," in *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pp. 79–90, 2012.

[5] Aji, A. M., Panwar, L. S., Ji, F., Chabbi, M., Murthy, K., Balaji, P., Bisset, K. R., Dinan, J., Feng, W., Mellor-Crummey, J. M., Ma, X., and Thakur, R., "On the efficacy of gpu-integrated MPI for scientific applications," in *The 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'13, New York, NY, USA - June 17 - 21, 2013*, pp. 191–202, 2013.

[6] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A., "Hedera: Dynamic flow scheduling for data center networks," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*, pp. 281–296, 2010.

[7] Alphabet, "Announcing gpus for google cloud platform." `https://goo.gl/Lv4p7m`, Nov. 2016.

[8] Alverson, R., Roweth, D., and Kaplan, L., "The gemini system interconnect," in *IEEE 18th Annual Symposium on High Performance Interconnects, HOTI 2010, Google Campus, Mountain View, California, USA, August 18-20, 2010*, pp. 83–87, 2010.

[9] Amazon, "New ec2 instance type the cluster gpu instance." `https://goo.gl/sYyKq5`, Nov. 2010.

[10] AMD, "Hardware-based scheduling of gpu work." `https://goo.gl/J6prg1`, Nov. 2012.

[11] AMD, "Amd unveils 6th generation a-series processor." `https://goo.gl/UZAmue`, June 2015.

[12] Amodei, D., Anubhai, R., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Chen, J., Chrzanowski, M., Coates, A., Diamos, G., Elsen, E., Engel, J., Fan, L., Fougner, C., Hannun, A. Y., Jun, B., Han, T., LeGresley, P., Li, X., Lin, L., Narang, S., Ng, A. Y., Ozair, S., Prenger, R., Qian, S., Raiman, J., Satheesh, S., Seetapun, D., Sengupta, S., Wang, C., Wang, Y., Wang, Z., Xiao, B., Xie, Y., Yogatama, D., Zhan, J., and Zhu, Z., "Deep speech 2 : End-to-end speech recognition in english and mandarin," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pp. 173–182, 2016.

[13] Bakhoda, A., Yuan, G. L., Fung, W. W. L., Wong, H., and Aamodt, T. M., "Analyzing CUDA workloads using a detailed GPU simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA, Proceedings*, pp. 163–174, 2009.

[14] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T. L., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pp. 164–177, 2003.

[15] Basaran, C. and Kang, K., "Supporting preemptive task executions and memory copies in gpgpus," in *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, pp. 287–296, 2012.

[16] Becchi, M., Sajjapongse, K., Graves, I., Procter, A. M., Ravi, V. T., and Chakradhar, S. T., "A virtual memory based runtime to support multi-tenancy in clusters with gpus," in *The 21st International Symposium on High-Performance Parallel and Distributed Computing, HPDC'12, Delft, Netherlands - June 18 - 22, 2012*, pp. 97–108, 2012.

[17] Belviranli, M. E., Bhuyan, L. N., and Gupta, R., "A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures," *TACO*, vol. 9, no. 4, p. 57, 2013.

[18] Bennett, J., Abbasi, H., Bremer, P., Grout, R. W., Gyulassy, A., Jin, T., Klasky, S., Kolla, H., Parashar, M., Pascucci, V., Pébay, P. P., Thompson, D. C., Yu, H., Zhang, F., and Chen, J., "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, p. 49, 2012.

[19] Boyer, M., Skadron, K., Che, S., and Jayasena, N., "Load balancing in a changing world: dealing with heterogeneity and performance variability," in *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013*, pp. 21:1–21:10, 2013.

[20] Braspenning, P. J., Thuijsman, F., and Weijters, A. J. M. M., eds., *Artificial Neural Networks: An Introduction to ANN Theory and Practice*, vol. 931 of *Lecture Notes in Computer Science*, Springer, 1995.

[21] Brin, S. and Page, L., "The anatomy of a large-scale hypertextual web search engine," *Computer Networks*, vol. 30, no. 1-7, pp. 107–117, 1998.

[22] Bussmann, M., Burau, H., Cowan, T. E., Debus, A., Huebl, A., Juckeland, G., Kluge, T., Nagel, W. E., Pausch, R., Schmitt, F., Schramm, U., Schuchart, J., and Widera, R., "Radiative signatures of the relativistic kelvin-helmholtz instability," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pp. 5:1–5:12, 2013.

[23] Chabbi, M., Murthy, K., Fagan, M. W., and Mellor-Crummey, J. M., "Effective sampling-driven performance tools for gpu-accelerated supercomputers," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pp. 43:1–43:12, 2013.

[24] Chintala, S., "Easy benchmarking of all publicly accessible implementations of convnets." `https://goo.gl/T25hRT`, Oct. 2016.

[25] Chowdhury, M. and Stoica, I., "Coflow: a networking abstraction for cluster applications," in *11th ACM Workshop on Hot Topics in Networks, HotNets-XI, Redmond, WA, USA - October 29 - 30, 2012*, pp. 31–36, 2012.

[26] Chowdhury, M., Zhong, Y., and Stoica, I., "Efficient coflow scheduling with varys," in *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pp. 443–454, 2014.

[27] Cielak, R., "Exploring neural networks, p.3." `https://goo.gl/YsXma9`, Feb. 2016.

[28] Cui, H., Zhang, H., Ganger, G. R., Gibbons, P. B., and Xing, E. P., "Geeps: scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pp. 4:1–4:16, 2016.

[29] Databricks, "Deep learning with apache spark and tensorflow." `https://goo.gl/n5795w`, Jan. 2016.

[30] Dayal, J., Lofstead, J. F., Eisenhauer, G., Schwan, K., Wolf, M., Abbasi, H., and Klasky, S., "SODA: science-driven orchestration of data analytics," in *11th IEEE International Conference on e-Science, e-Science 2015, Munich, Germany, August 31 - September 4, 2015*, pp. 475–484, 2015.

[31] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A. W., Tucker, P. A., Yang, K., and Ng, A. Y., "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, pp. 1232–1240, 2012.

[32] Dean, J. and Ghemawat, S., "Mapreduce: Simplified data processing on large clusters," in *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pp. 137–150, 2004.

[33] Deng, J., Dong, W., Socher, R., Li, L., Li, K., and Li, F., "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*, pp. 248–255, 2009.

[34] Dettmers, T., "How to parallelize deep learning on gpus part 1/2: Data parallelism." `https://goo.gl/4GK6AP`, Oct. 2014.

[35] Dettmers, T., "How to parallelize deep learning on gpus part 2/2: Model parallelism." `https://goo.gl/PF2W8C`, Nov. 2014.

[36] Diamos, G., Sengupta, S., Catanzaro, B., Chrzanowski, M., Coates, A., Elsen, E., Engel, J., Hannun, A. Y., and Satheesh, S., "Persistent rnns: Stashing recurrent weights on-chip," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pp. 2024–2033, 2016.

[37] DoE, "Top ten exascale research challenges." `https://goo.gl/Pyr7Ms`, Feb. 2014.

[38] Duato, J., Peña, A. J., Silla, F., Mayo, R., and Quintana-Ortí, E. S., "rcuda: Reducing the number of gpu-based accelerators in high performance clusters," in *Proceedings of the 2010 International Conference on High Performance Computing & Simulation, HPCS 2010, June 28 - July 2, 2010, Caen, France*, pp. 224–231, 2010.

[39] Funahashi, K. and Nakamura, Y., "Approximation of dynamical systems by continuous time recurrent neural networks," *Neural Networks*, vol. 6, no. 6, pp. 801–806, 1993.

[40] Gamell, M., Rodero, I., Parashar, M., Bennett, J., Kolla, H., Chen, J., Bremer, P., Landge, A. G., Gyulassy, A., McCormick, P. S., Pakin, S., Pascucci, V., and Klasky, S., "Exploring power behaviors and trade-offs of in-situ data analytics," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pp. 77:1–77:12, 2013.

[41] Giunta, G., Montella, R., Agrillo, G., and Coviello, G., "A GPGPU transparent virtualization component for high performance computing clouds," in *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part I*, pp. 379–391, 2010.

[42] Goswami, A., Tian, Y., Schwan, K., Zheng, F., Young, J., Wolf, M., Eisenhauer, G., and Klasky, S., "Landrush: Rethinking in-situ analysis for GPGPU workflows," in *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016*, pp. 32–41, 2016.

[43] Goswami, A., Young, J., Schwan, K., Farooqui, N., Gavrilovska, A., Wolf, M., and Eisenhauer, G., "Gpushare: Fair-sharing middleware for GPU clouds," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*, pp. 1769–1776, 2016.

[44] Greenberg, A. G., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S., "VL2: a scalable and flexible data center network," in *Proceedings of the ACM SIGCOMM 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Barcelona, Spain, August 16-21, 2009*, pp. 51–62, 2009.

[45] Gregg, C., Dorn, J., Hazelwood, K. M., and Skadron, K., "Fine-grained resource sharing for concurrent GPGPU kernels," in *4th USENIX Workshop on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 7-8, 2012*, 2012.

[46] Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., and Ranganathan, P., "Gvim: Gpu-accelerated virtual machines," in *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, (New York, NY, USA), pp. 17–24, ACM, 2009.

[47] Gupta, V., Schwan, K., Tolia, N., Talwar, V., and Ranganathan, P., "Pegasus: Coordinated scheduling for virtualized accelerator-based systems," in *2011 USENIX Annual Technical Conference, Portland, OR, USA, June 15-17, 2011*, 2011.

[48] He, K., Zhang, X., Ren, S., and Sun, J., "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[49] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I., "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, 2011.

[50] Houben, S., Stallkamp, J., Salmen, J., Schlipsing, M., and Igel, C., "Detection of traffic signs in real-world images: The german traffic sign detection benchmark," in *The 2013 International Joint Conference on Neural Networks, IJCNN 2013, Dallas, TX, USA, August 4-9, 2013*, pp. 1–8, 2013.

[51] hsafoundation.com, "Heterogeneous systems architecture - standards." `https://goo.gl/dcx5OL`, Aug. 2012.

[52] Iandola, F. N., Ashraf, K., Moskewicz, M. W., and Keutzer, K., "Firecaffe: near-linear acceleration of deep neural network training on compute clusters," *CoRR*, vol. abs/1511.00175, 2015.

[53] IBM, "Ibm cloud first to offer latest nvidia grid with tesla m60 gpu." `https://goo.gl/Lv4p7m`, May 2016.

[54] Ibrahim, K. Z., Madduri, K., Williams, S., Wang, B., Ethier, S., and Oliker, L., "Analysis and optimization of gyrokinetic toroidal simulations on homogenous and heterogenous platforms," *IJHPCA*, vol. 27, no. 4, pp. 454–473, 2013.

[55] ICL-UTK, "Magma - matrix algebra on gpu and multicore architectures." `https://goo.gl/o22iCp`, Aug. 2016.

[56] Intel, "Gen8 compute architecture of intel graphics." `https://goo.gl/9GbeJT`, July 2015.

[57] Intel, "Intel math kernel library benchmarks." https://goo.gl/cVC7pb, Sept. 2015.

[58] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R. B., Guadarrama, S., and Darrell, T., "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, November 03 - 07, 2014*, pp. 675–678, 2014.

[59] Johnson, J., "Benchmarks for popular cnn models." https://goo.gl/3dzjxr, Oct. 2016.

[60] Kato, S., Lakshmanan, K., Kumar, A., Kelkar, M., Ishikawa, Y., and Rajkumar, R., "RGEM: A responsive GPGPU execution model for runtime engines," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria, November 29 - December 2, 2011*, pp. 57–66, 2011.

[61] Kato, S., Lakshmanan, K., Rajkumar, R., and Ishikawa, Y., "Timegraph: GPU scheduling for real-time multi-tasking environments," in *2011 USENIX Annual Technical Conference, Portland, OR, USA, June 15-17, 2011*, 2011.

[62] Kato, S., McThrow, M., Maltzahn, C., and Brandt, S. A., "Gdev: First-class GPU resource management in the operating system," in *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pp. 401–412, 2012.

[63] KhronosGroup, "Opencl - the open standard for parallel programming of heterogeneous systems." https://goo.gl/dAuzD9, Dec. 2008.

[64] KhronosGroup, "Opencl 2.0 device enqueue." http://goo.gl/NRTwqn, Nov. 2014.

[65] Klug, T., Ott, M., Weidendorfer, J., and Trinitis, C., "autopin - automated optimization of thread-to-core pinning on multicore systems," *Trans. HiPEAC*, vol. 3, pp. 219–235, 2011.

[66] Krieder, S. J., Wozniak, J. M., Armstrong, T. G., Wilde, M., Katz, D. S., Grimmer, B., Foster, I. T., and Raicu, I., "Design and evaluation of the gemtc framework for gpu-enabled many-task computing," in *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, pp. 153–164, 2014.

[67] Krizhevsky, A., Sutskever, I., and Hinton, G. E., "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, pp. 1106–1114, 2012.

[68] Lee, J., Samadi, M., Park, Y., and Mahlke, S. A., "Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*, pp. 245–255, 2013.

[69] Leskovec, J. and Faloutsos, C., "Scalable modeling of real graphs using kronecker multiplication," in *Machine Learning, Proceedings of the Twenty-Fourth International Conference (ICML 2007), Corvallis, Oregon, USA, June 20-24, 2007*, pp. 497–504, 2007.

[70] Leskovec, J. and Sosic, R., "SNAP: A general-purpose network analysis and graph-mining library," *ACM TIST*, vol. 8, no. 1, p. 1, 2016.

[71] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B., "Scaling distributed machine learning with the parameter server," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pp. 583–598, 2014.

[72] Li, M., Zhang, T., Chen, Y., and Smola, A. J., "Efficient mini-batch training for stochastic optimization," in *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pp. 661–670, 2014.

[73] Li, T., Brett, P., Knauerhase, R. C., Koufaty, D. A., Reddy, D., and Hahn, S., "Operating system support for overlapping-isa heterogeneous multi-core architectures," in *16th International Conference on High-Performance Computer Architecture (HPCA-16 2010), 9-14 January 2010, Bangalore, India*, pp. 1–12, 2010.

[74] Liu, C. L. and Layland, J. W., "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[75] LLNL, "Sierra advanced technology system." `https://goo.gl/Lzrka5`, Nov. 2014.

[76] Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G., "Pregel: a system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pp. 135–146, 2010.

[77] Mars, J., Tang, L., Hundt, R., Skadron, K., and Soffa, M. L., "Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations," in *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, 3-7 December 2011, Porto Alegre, Brazil*, pp. 248–259, 2011.

[78] Mars, J., Vachharajani, N., Hundt, R., and Soffa, M. L., "Contention aware execution: online contention detection and response," in *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*, pp. 257–265, 2010.

[79] Martinasso, M., Kwasniewski, G., Alam, S. R., Schulthess, T. C., and Hoefler, T., "A pcie congestion-aware performance model for densely populated accelerator servers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, p. 63, 2016.

[80] Massoulié, L. and Roberts, J., "Bandwidth sharing: Objectives and algorithms," in *Proceedings IEEE INFOCOM '99, The Conference on Computer Communications, Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies, The Future Is Now, New York, NY, USA, March 21-25, 1999*, pp. 1395–1403, 1999.

[81] MaximMilakov, "nnforge : Convolutional and fully-connected neural networks c++ framework." https://goo.gl/9Nuga3, July 2016.

[82] Menychtas, K., Shen, K., and Scott, M. L., "Disengaged scheduling for fair, protected access to fast computational accelerators," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pp. 301–316, 2014.

[83] Microsoft, "New g2 instance type with 4x more gpu power." https://goo.gl/qKYH6E, Apr. 2015.

[84] Microsoft, "Azure n-series preview availability." https://goo.gl/ar7ehL, Aug. 2016.

[85] nanomsg.org, "nanomsg library." https://goo.gl/vD1eFb, Nov. 2014.

[86] Nervanasys, "Neon - the worlds fastest deep learning framework." https://goo.gl/rgC9XR, Sept. 2016.

[87] nextplatform.com, "Inside the gpu clusters that power baidu's neural networks." https://goo.gl/DqA8ab, Dec. 2015.

[88] Nvidia, "Nvidia gpus - the engine of deep learning." https://goo.gl/6SmcYO.

[89] Nvidia, "About cuda." https://goo.gl/og3jYS, Mar. 2006.

[90] Nvidia, "Nvidia gpus power adobe creative cloud." https://goo.gl/4lvpok, Nov. 2006.

[91] Nvidia, "Nvidia unveils cuda." https://goo.gl/n5qAP4, Nov. 2006.

[92] Nvidia, "Kepler architecture." https://goo.gl/aY3Bfg, Apr. 2012.

[93] Nvidia, "An introduction to cuda-aware mpi." https://goo.gl/h7zg0d, Mar. 2013.

[94] Nvidia, "Cuda dynamic parallelism." https://goo.gl/TBIuVh, May 2014.

[95] Nvidia, "What is nvlink?." https://goo.gl/CeGD6v, Nov. 2014.

[96] Nvidia, "Cuda 7 performance report." http://bit.ly/2fZMr1w, May 2015.

[97] Nvidia, "Cuda multi-process service." https://goo.gl/sSydPk, Mar. 2015.

[98] Nvidia, "cublas - gpu accelerated basic linear algebra subroutines." https://goo.gl/Al2IRy, Apr. 2016.

[99] Nvidia, "Cuda profiling tools interface." https://goo.gl/dq0nao, Apr. 2016.

[100] Nvidia, "cudnn - gpu accelerated deep learning." https://goo.gl/WNP3bF, Apr. 2016.

[101] Nvidia, "cufft - gpu accelerated fast fourier transform." https://goo.gl/YopE0o, Apr. 2016.

[102] Nvidia, "Gpu applications catalog." https://goo.gl/Kfr5Bb, Nov. 2016.

[103] NVIDIA, "Nvidia dgx saturnv ranked worlds most efficient supercomputer by wide margin." `https://goo.gl/1pz9sa`, Nov. 2016.

[104] NVIDIA, "Nvidia pascal gpus to double speed of europe's fastest supercomputer." https://goo.gl/yj0dMZ, Apr. 2016.

[105] OLCF, "Introducing titan - advancing the era of accelerated computing." `https://goo.gl/0cqbOz`, Nov. 2012.

[106] OLCF, "Summit the next peak in hpc." `https://goo.gl/pikrTW`, Nov. 2014.

[107] OPENACC.ORG, "Openacc - directives for accelerators." `https://goo.gl/UAAiyd`, Nov. 2011.

[108] PAI, S., THAZHUTHAVEETIL, M. J., and GOVINDARAJAN, R., "Improving GPGPU concurrency with elastic kernels," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pp. 407–418, 2013.

[109] PARK, J. J. K., PARK, Y., and MAHLKE, S. A., "Chimera: Collaborative preemption for multitasking on a shared GPU," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pp. 593–606, 2015.

[110] PHULL, R., LI, C., RAO, K., CADAMBI, S., and CHAKRADHAR, S. T., "Interference-driven resource management for gpu-based heterogeneous clusters," in *The 21st International Symposium on High-Performance Parallel and Distributed Computing, HPDC'12, Delft, Netherlands - June 18 - 22, 2012*, pp. 109–120, 2012.

[111] PLIMPTON, S., "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.*, vol. 117, pp. 1–19, Mar. 1995.

[112] QIU, Z., STEIN, C., and ZHONG, Y., "Minimizing the total weighted completion time of coflows in datacenter networks," in *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, pp. 294–303, 2015.

[113] RAVI, V. T., BECCHI, M., AGRAWAL, G., and CHAKRADHAR, S. T., "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," in *Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing, HPDC 2011, San Jose, CA, USA, June 8-11, 2011*, pp. 217–228, 2011.

[114] ROSSBACH, C. J., CURREY, J., SILBERSTEIN, M., RAY, B., and WITCHEL, E., "Ptask: operating system abstractions to manage gpus as compute devices," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pp. 233–248, 2011.

[115] ROSSBACH, C. J., YU, Y., CURREY, J., MARTIN, J., and FETTERLY, D., "Dandelion: a compiler and runtime for heterogeneous systems," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pp. 49–68, 2013.

[116] Sajjapongse, K., Wang, X., and Becchi, M., "A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with gpus," in *The 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'13, New York, NY, USA - June 17 - 21, 2013*, pp. 179–190, 2013.

[117] Seide, F. and Agarwal, A., "CNTK: microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, p. 2135, 2016.

[118] Sengupta, D., Goswami, A., Schwan, K., and Pallavi, K., "Scheduling multi-tenant cloud workloads on accelerator-based systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, pp. 513–524, 2014.

[119] Seo, S., Yoon, E. J., Kim, J., Jin, S., Kim, J., and Maeng, S., "HAMA: an efficient matrix computation with the mapreduce framework," in *Cloud Computing, Second International Conference, CloudCom 2010, November 30 - December 3, 2010, Indianapolis, Indiana, USA, Proceedings*, pp. 721–726, 2010.

[120] Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., and LeCun, Y., "Overfeat: Integrated recognition, localization and detection using convolutional networks," *CoRR*, vol. abs/1312.6229, 2013.

[121] Shi, L., Chen, H., and Sun, J., "vcuda: GPU accelerated high performance computing in virtual machines," in *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pp. 1–11, 2009.

[122] Simonyan, K. and Zisserman, A., "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[123] skymind.io, "Distributed deep learning, part 1: An introduction to distributed training of neural networks." `https://goo.gl/VNZKlL`, Oct. 2016.

[124] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A. C., and Peterson, L. L., "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, pp. 275–287, 2007.

[125] Stephenson, M., Hari, S. K. S., Lee, Y., Ebrahimi, E., Johnson, D. R., Nellans, D. W., O'Connor, M., and Keckler, S. W., "Flexible software profiling of GPU architectures," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pp. 185–197, 2015.

[126] Stratton, J. A., Stone, S. S., and Hwu, W. W., "MCUDA: an efficient implementation of CUDA kernels for multi-core cpus," in *Languages and Compilers for Parallel Computing, 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers*, pp. 16–30, 2008.

[127] Suzuki, Y., Kato, S., Yamada, H., and Kono, K., "Gpuvm: Why not virtualizing gpus at the hypervisor?," in *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pp. 109–120, 2014.

[128] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A., "Going deeper with convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pp. 1–9, 2015.

[129] Tanasic, I., Gelado, I., Cabezas, J., Ramírez, A., Navarro, N., and Valero, M., "Enabling preemptive multiprogramming on gpus," in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pp. 193–204, 2014.

[130] Tang, L., Mars, J., and Soffa, M. L., "Compiling for niceness: mitigating contention for qos in warehouse scale computers," in *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*, pp. 1–12, 2012.

[131] Tang, L., Mars, J., Wang, W., Dey, T., and Soffa, M. L., "Reqos: reactive static/dynamic compilation for qos in warehouse scale computers," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pp. 89–100, 2013.

[132] Tian, K., Dong, Y., and Cowperthwaite, D., "A full GPU virtualization solution with mediated pass-through," in *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pp. 121–132, 2014.

[133] Tian, Y., Klasky, S., Yu, W., Abbasi, H., Wang, B., Podhorszki, N., Grout, R. W., and Wolf, M., "A system-aware optimized data organization for efficient scientific analytics," in *The 21st International Symposium on High-Performance Parallel and Distributed Computing, HPDC'12, Delft, Netherlands - June 18 - 22, 2012*, pp. 125–126, 2012.

[134] UCDavis, "Gunrock: High-performance graph primitives for the gpu." `https://goo.gl/GZQyWS`, Nov. 2016.

[135] Valiant, L. G., "Why BSP computers?," in *The Seventh International Parallel Processing Symposium, Proceedings, Newport Beach, California, USA, April 13-16, 1993.*, pp. 2–5, 1993.

[136] Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S., and LeCun, Y., "Fast convolutional nets with fbfft: A GPU performance evaluation," *CoRR*, vol. abs/1412.7580, 2014.

[137] Wang, B., Ethier, S., Tang, W. M., Williams, T. J., Ibrahim, K. Z., Madduri, K., Williams, S., and Oliker, L., "Kinetic turbulence simulations at extreme scale on leadership-class systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pp. 82:1–82:12, 2013.

[138] Wehrle, K., Güneş, M., and Gross, J., eds., *Modeling and Tools for Network Simulation*. Springer, 2010.

[139] Yair, E. and Gersho, A., "The boltzmann perceptron network: A multi-layered feed-forward network equivalent to the boltzmann machine," in *Advances in Neural*

Information Processing Systems 1, [NIPS Conference, Denver, Colorado, USA, 1988], pp. 116–123, 1988.

[140] Yalamanchili, P. and others, "ArrayFire - A high performance software library for parallel computing with an easy-to-use API," 2015.

[141] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pp. 15–28, 2012.

[142] Zhao, Y., Chen, K., Bai, W., Yu, M., Tian, C., Geng, Y., Zhang, Y., Li, D., and Wang, S., "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *2015 IEEE Conference on Computer Communications, INFOCOM 2015, Kowloon, Hong Kong, April 26 - May 1, 2015*, pp. 424–432, 2015.

[143] Zheng, F., Yu, H., Hantas, C., Wolf, M., Eisenhauer, G., Schwan, K., Abbasi, H., and Klasky, S., "Goldrush: resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pp. 78:1–78:12, 2013.

[144] Zheng, F., Zou, H., Eisenhauer, G., Schwan, K., Wolf, M., Dayal, J., Nguyen, T., Cao, J., Abbasi, H., Klasky, S., Podhorszki, N., and Yu, H., "Flexio: I/O middleware for location-flexible scientific data analytics," in *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*, pp. 320–331, 2013.

[145] Zhou, A. C., He, B., Cheng, X., and Lau, C. T., "A declarative optimization engine for resource provisioning of scientific workflows in iaas clouds," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2015, Portland, OR, USA, June 15-19, 2015*, pp. 223–234, 2015.

[146] Zinkevich, M., Weimer, M., Smola, A. J., and Li, L., "Parallelized stochastic gradient descent," in *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010, Vancouver, British Columbia, Canada.*, pp. 2595–2603, 2010.

VITA

Anshuman Goswami was born and raised at Serampore, a suburb of Kolkata, India. He went to Jadavpur University for his undergraduate studies majoring in Electronics and Telecommunication Engineering in 2005. He worked in the industry as a software engineer for the next 5 years, first, in Tata Consultancy Services Ltd and then, in STMircoelectronics NV. He came back to school to pursue a PhD in Computer Science from Georgia Institute of Technology in 2010. He is currently a PhD candidate at the School of Computer Science in the College of Computing at Georgia Tech being advised by Prof Karsten Schwan and Dr Matthew Wolf.